DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A FLOWCHARTING SYSTEM AND COMPILER INTERFACE FOR MACPITTS

by

Elbert L. Weist Jr.

June 1986

Co Advisors: H.H. Loomis
D.E. Kirk

Approved for public release; distribution is unlimited.

T233049

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION NCLASSIFIED	1b RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION: aval Postgraduate School 6b OFFICE SYMBOL (If applicable): 62	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
ADDRESS (City, State, and ZIP Code): onterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
FLOWCHARTING SYSTEM AND COMPILER INTERFACE FOR MACPITTS

PERSONAL AUTHOR(S)
bert L. Weist Jr.

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ster's Thesis	FROM _____ TO _____	86 June 20	189

SUPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Flowcharting; MACPITTS; SCALD; AWK

ABSTRACT (Continue on reverse if necessary and identify by block number)

The feasibility of a direct FLOWCHART-TO-CHIP concept for the MACPITTS silicon compiler is demonstrated. A graphics package is developed that contains all flowcharting symbols and interconnection systems required for implementation of a MACPITTS compiled finite state machine. This provides the capability for the user's input to MACPITTS to be a graphically generated finite state machine algorithmic flowchart instead of a complex LISP language input file. The SCALD system developed by the VALID corporation provides the required CAD tools. A flowchart to MACPITTS compiler is also developed that demonstrates the validity of the flowchart input concept. Complex flowcharts have been created and successfully compiled into the correct LISP language input file format required by MACPITTS.

DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
NAME OF RESPONSIBLE INDIVIDUAL of H. H. Loomis	22b TELEPHONE (Include Area Code) (408)646-3214 22c OFFICE SYMBOL 62Lm

FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release, distribution unlimited

A Flowcharting System and Compiler Interface for MACPITTS

by

Elbert L. Weist Jr.
Major, United States Marine Corps
B. S., California State Polytechnic University, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1986

ABSTRACT

The feasibility of a direct FLOWCHART-TO-CHIP concept for the MACPITTS silicon compiler is demonstrated. A graphics package is developed that contains all flowcharting symbols and interconnection systems required for implementation of a MACPITTS compiled finite state machine. This provides the capability for the user's input to MACPITTS to be a graphically generated finite state machine algorithmic flowchart instead of a complex LISP language input file. The SCALD system developed by the VALID corporation provides the required CAD tools. A flowchart to MACPITTS compiler is also developed that demonstrates the validity of the flowchart input concept. Complex flowcharts have been created and successfully compiled into the correct LISP language input file format required by MACPITTS.

TABLE OF CONTENTS

I. INTRODUCTION 8

A. BACKGROUND 8

B. GOALS 8

C. IMPLEMENTATION 9

II. MACPITTS BASICS 10

A. MACPITTS INTRODUCTION 10

B. MACPITTS INPUT FILE FORMAT 10

1. Program Name 10

2. System Definitions 11

a. Power Ground and Clock Pads 11

b. Signals 12

c. Registers and Flags 14

3. Processes 15

4. Circuit Description Block 16

C. CIRCUIT PARAMETERS 16

1. States 16

2. Conditionals 19

3. Action statements 20

4. Detailed Example Explanations 21

a. Forced Parallel Execution 24

b. Serial Constructs 24

c. Serial-Parallel 26

5. Reset and Always Functions 26

D. MACPITTS INTERPRETER 28

III. FLOWCHART INTERFACE THEORY 30

A. SYMBOLS 31

1. State 31

2. Conditional 33

3. Intrastate 33

4. Title and Process Block 34

5. Signalnames 34

B. SPECIFIC RULES OF THIS FLOWCHART SYSTEM 37

1. Title and Process Rules 37

2. MACPITTS Required Definitions 38

a. Signals 39

b. Ports 40

c. Registers and Flags 41

d. Constants and Power 41

C. STATES 42

1. Flowchart Implementation 42

2. State Content 43

3. Symbol Differences 44

D. Conditionals 44

1. Conditional Test Capabilities 48

2. Symbol Configuration 50

E. INTRASTATE 50

IV. SCALD SYSTEM 53

A. SCALD CAPABILITY OVERVIEW 53

B. THE SCALD GRAPHICS EDITOR 57

1. Screen Functions 59

2. Puck or Mouse Use 60

a. Yellow 60

b. Blue 60

c. Green 61

d. White 61

3. Logic Versus Body Drawings 61

4. Creating New Bodies and Versions of Bodies 62

a. The Body 63

b. Versions 65

5. Creating Logic Drawings 68

6. Signalname Change Commands 70

C. THE CONNECTIVITY FILE 71

V. FLOWCHART COMPILER 73

A. CONCEPT and METHOD OF ATTACK 73

B. AWK 76

1. General AWK Concept 76

a. Case Basis 76

b. Arrays 81

c. Action Statements 81

d. Other Useful AWK Commands and Variables 83

C. DETAILED COMPILER OVERVIEW 84

D. COMPILER SPECIFICS 85

1. Doit 85

2. Filenamechange 86

3. First 86

4. Thesisprop 87

5. Sigsort 88

6. Finalsort 90

7. Thesisdef 96

E. LIMITATIONS 97

1. Function Syntax 99

2. Maximum Number of Signals Allowed 100

3. Conditional Case Evaluation Limitations 100

4. MACPITTS Subroutine Implementation 101

5. Single Process Computation 102

6. Serial Action Implementation 102

7. SCALD System Format Requirement 103

F. COMPILER SUMMARY 103

VI. CONCLUSIONS 105

A. SUMMARY 105

B. RECOMMENDATIONS 106

VII. APPENDIX A 107

VIII. APPENDIX B 136

I. INTRODUCTION

A. BACKGROUND

The art of silicon compilation is still struggling in its infancy and has more than its share of detractors. Indeed, the currently available silicon compilers have many drawbacks, some of which may never be adequately solved. However, one common factor among all of these compilers is their compiler specific input language. As an example, the MACPITTS compiler requires a LISP language type of input file. Any user must learn this specific input language and know it well to develop even a relatively simple chip. Any change to another compiler will require a concommitant expenditure of energy in learning the new input language.

B. GOALS

This thesis introduces the concept of direct Flowchart-To-Chip silicon compilation. It is based on the premise that any engineer can, with relative ease, create a graphic flowchart that fully describes a desired circuit. Little understanding of the actual silicon compiler, other than its capabilities and limitations, is required. Learning a new input language when shifting from one compiler to another would not be required.

To demonstrate the feasability of this concept a functional flowchart compiler

capable of converting a flowchart into a syntactically correct MACPITTS input file has been developed. MACPITTS must then be called upon to generate the chip layout based on current vlsi fabrication technology.

Applications of this flowchart compiler could conceivably be extended beyond the silicon compiler field to the more general program design field. It becomes a machine and language independent "language" just as it has always been.

C. IMPLEMENTATION

The next four chapters discuss the MACPITTS silicon compiler and its required input file, the flowcharting method conceived during this thesis, the graphics work station used to implement this flowchart, and the compiler written to convert the flowchart into a MACPITTS input file. Detailed examples are used, whenever complex ideas or computer code are presented, to enhance the reader's understanding of the material.

II. MACPITTS BASICS

A. MACPITTS INTRODUCTION

MACPITTS is a silicon compiler which creates VLSI circuits based on a finite state machine implementation. It is written in the LISP language [Ref. 6] and requires a LISP type input file description of the desired circuit. A detailed discussion of this input file is included in later sections of this chapter.

MACPITTS can be extremely complex and several weeks of concentrated effort may be required to master its nuances. However, in an attempt to simplify this task the following sections will contain numerous examples of MACPITTS input file code and detailed explanations.

B. MACPITTS INPUT FILE FORMAT

Every MACPITTS input file contains four major blocks of data. Some are only one line long while others are detailed descriptions of complex sub-circuits. Each block is covered below.

1. Program Name

The first line of the input file (hereafter referred to as the file) contains the user's designated title and a decimal integer designating the maximum width of the data path. All registers, input and output ports, and some of the data path manipulation elements will be as wide as this number indicates. There is

currently no way to change the width of individual registers. Therefore, the size of the chip is dramatically influenced by the largest required register. The program name line syntax is:

(program <program_name> <data_path_width_integer>

where the leading parenthesis is part of the LISP language requirement. Its closing match is at the end of file.

2. System Definitions

The next block of code in the file contains the definitions of the "variables" in the circuit. These include the power, ground and clock pad definitions, the register and flag definitions, and the signal and port definitions.

a. Power Ground and Clock Pads

Maximum separation of the power and ground pads on the chip is guaranteed because only three sides of the chip are used for pads. The ground pad is normally designated as pad number one and the power pad is designated with the highest pad number on the chip.

Currently MACPITTS requires a three-phase clock supplied externally. Therefore, three clock pads must also be designated and are usually given pad numbers two, three and four. The following is an example of these definitions based on a ten pad chip:

```
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def 10 power)
```

The syntax is (def <pad_number> <special_pad_name>). The order of the pad number and name should be noted as these are the only pads designated in this manner. All others have any required pad number (or numbers) entered as the last entry in the syntax.

b. Signals

Signal and port pad definitions are basically the same, yet there are many subtle circuit differences between them. There are four types of each: input or output, input and output (I/O), tristate, and internal. The syntax is:

(def <signal_name> signal <type> [<pad_number>])

(def <port_name> port <type> [(pad_numbers)])

The standard input or output only type requires a pad number for a signal and sequential ascending decimal integers for a port. The number of port pads must equal the data path width defined in the program title line. The port pad numbers are enclosed in parenthesis and each is separated by a single space. Examples:

```
(def A signal input 5)
(def a signal output 6)
(def B port input (7 8 9 10))          a 4 bit wide data path
(def b port output (11 12 13 14 15))   a 5 bit wide data path
```

When a signal is set in the process block using the setq function of LISP, it is set to a boolean value of true (t). A 1 or 0 will not be accepted as a valid signal value. A port, however, is set to an integer value. In either case, these values are only valid during the state in which they are set. In all other states, MACPITTS

sets them to false (f) or if a port to 0. Therefore, setting a signal to false is a useless command. MACPITTS has already set it to that value. Also, signals and ports cannot change value more than once in a single state. To attempt to do so will cause circuit oscillation.

The input/output (I/O) type requires the same definition format as the input only type with the type changed, e.g. (def A port i/o (6 7 8)). It should be noted that all types are in lower case text notation. The circuit differences are more complex for this I/O configuration. The pads must be set up explicitly for their input or output function. In the signal case a 1 or 0 is now a valid set condition in that it designates the input or output function of the pad (1 for input and 0 for output). As noted before, the value of the signal must be boolean (true) and its value will be valid only during the state in which it is set. However, the I/O direction doesn't change until it is explicitly set to its opposite direction. For the port there is a MACPITTS documentation problem. Since the numbers 0 and 1 are valid integers there is no way to designate the I/O direction that won't be interpreted as an output value. One could assume that using a boolean value in setting a port could designate the I/O direction, but this has not been confirmed.

The tristate type is like the output only type. The two differences are the change in type name in the syntax and the "normal" MACPITTS" output value. Examples of the syntax are:

(def A signal tristate 5)
(def B port tristate (6 7 8 9))

The output value must be explicitly set to a true (t) or false (f) for a signal and to an explicit integer (including in this case the integer 0) for a port. MACPITTS sets the value of each to a high impedance state during all states in which they have not been explicitly set.

The last type is the internal signal or port. There are no pads involved in this type, therefore, there are no pad numbers required. Instead it designates a common point in the data or signal bus where one or more connections can be made internally in a multiplexed mode. A syntax example is:

(def A signal internal)
(def B port internal)

Again, this signal or port value is valid only during the state in which it is set.

c. Registers and Flags

Registers and flags require relatively simple definitions as shown by their syntax:

(def <register_name> register)
(def <flag_name> flag)

A register is an integer storage device that retains its set value in binary form until explicitly changed. Its size, as noted above, is the same as the data path size declared in the program title line. A flag is a single bit register that stores a boolean value (t or f) until it is explicitly changed. Both are implemented as master-slave flip flops and the value loaded during the action statement of a state appears at the output of the register at the end of the state clock cycle. It holds its old value throughout the current state. This is different from the signals and

ports which become active instantly when set and remain active only during the current state. Use of a signal rather than a flag to activate a conditional in a state located in another process (running in parallel with the current process) will cause different timing effects. Actuation of that state by a signal will occur during the current clock cycle, while activation by a flag will occur during the next clock cycle.

3. Processes

A process is in essence a separate and distinct MACPITTS state machine that runs in parallel with other processes in the same program (on the same chip). Interprocess communication is accomplished through signals, flags or registers. It should be noted that signal, port and register collisions that occur when two or more processes set the same signal, port or register during the same clock cycle will result in the OR of the collision values. This result only occurs when states from different processes collide. Signals and ports cannot collide within a single state without oscillation. This is described in chapter I and is forbidden in MACPITTS.

In any process only one state may be active during a single complete clock period of the three-phase clock. Additionally, this process is defined by a single line located at the beginning of the circuit description block of code which describes the states contained in that process. The syntax and an example are:

(process <process_name> <subroutine_stack_depth_integer>
Example:
(process First_process 0

Again, the single parenthesis should be noted. The closing parenthesis is located at the end of the circuit description code contained in that process. The stack depth is a user determined decimal integer that indicates the maximum depth of a first in last out subroutine return address stack. If no subroutines are used then this value should be set to 0.

4. Circuit Description Block

The circuit description is the heart of the MACPITTS input file. It contains individual states designated by state labels or names. Each state can contain conditional tests and actions that operate under a strict yet complex set of rules. These rules will be described in detail in the circuit parameters section of this chapter.

This description can also contain subroutines, however, the documentation available on MACPITTS subroutines is currently deficient enough so as to preclude a clear presentation in this document. Further study and analysis of this capability is deferred to another time.

C. CIRCUIT PARAMETERS

This section will present the MACPITTS circuit description code. The parallel and serial interactions of states, conditional tests, actions, and state transitions can become extremely complex. Therefore, each of these four will be covered in detail.

1. States

Each state is best described as the classically defined "state" in a finite state machine (FSM) controller. Many states may exist within a process, however, only one state may be active during a given clock cycle. In MACPITTS, as in a normal FSM, all operations that are contained in a state must be completed in one complete clock cycle. Based on propagation delays, the maximum clock speed depends upon the number of tests and actions that must be performed to complete the state. The slowest state will set the requirement for the fastest usable clock.

States within a given process cannot operate in parallel, however, states that exist in separate processes do run in parallel. Transitions from one state to another can occur in three different ways, but direct transitions may only occur within the same process. Transitions outside a process are forbidden, however, communication through signals etc. between processes is allowed. The three types of transitions are:

(go <designated_state_label>) "a forced transition"
"fall through"
"auto recycle"

Figure 2.1 provides examples of each of these methods.

The standard transition is the GO statement which forces a transition to the named state at the end of the current clock cycle. This is the method used in the Flowchart-to-Chip system developed in this thesis. If a state action block has no transition statement then MACPITTS automatically "falls through" to the

next state listed in the process. Finally, to prevent chip lockup, if the last state in the process has no transition statement, then MACPITTS automatically cycles back to the beginning of the last state. Therefore, if this state is ever reached it effectively becomes the only state in that process until the chip is reset.

In Figure 2.1, state1 uses a standard "go" transition to state2 or state3 depending upon the outcome of the conditional test of testsig1. The boolean

---

```
VARIABLE DEFINITIONS:
SIGNALS: testsig1, testsig2, signalout
PORTS: portout
REGISTERS: C, D
FLAGS: B

(process transition_methods 0
state1
        (cond (testsig1 (setq B t)(go state2))
              (t (go state3)))
state2
        (cond (testsig2 (setq C D)(setq D C))
              (t (setq C 3)))
state3
        ((setq signalout t)
         (setq portout C)
         (setq portout D))
)
(process Second 0
        .
        .
```

Figure 2.1 State Transition and Parallel/Serial Setq Action Example

---

result of that test is sent to the MACPITTS controller. Conditionals will be covered in detail shortly. State2 uses the "fall through" technique and transitions to state3 regardless of the results of the conditional test. Finally, state3 is an example of serial action operations and it obeys the "auto recycle" method of transition. Once state3 is reached in this example, transition out of it is impossible unless the chip is reset. Reset automatically resets the MACPITTS controller to the first state of each process and resets all registers and flags. One should also note that state3 cannot fall through to the next state because it is in a different process.

2. Conditionals

Conditionals comprise the most complex constructs available in MACPITTS. Based on parallel operation, they can be formed into multiple serial or case basis test systems. Each test can be a single boolean test or a complex function that returns a boolean value. ALL tests and actions in a conditional must be completed in one clock cycle, hence, the use of parallel evaluation and action implementations.

The syntax of a conditional is (see Figure 2):

(cond ((<function_tested_for_boolean_true>)(<action1>)(<actionN>)(<transition>))

(else_always_true_t(action1)(actionN)(transition))

The function tested for a boolean true can be as complex as desired based on the available MACPITTS functions listed in chapter III or as simple as the boolean value "t". A boolean value "t" always tests true. The action can be a setq

(discussed shortly) or another conditional test with its own actions and transitions. This construct will be referred to as a serial conditional in this document. It is important to realize that all the actions and transitions in a given conditional occur in parallel. This reduces the propagation delay times, but also causes chip size to increase dramatically and, therefore, commands caution in development of action statements. Figure 2.2 provides specific examples of the various types of conditional constructs. A detailed explanation of these examples will be presented after the SETQ action statement has been introduced.

3. Action statements

All MACPITTS actions are accomplished through the LISP "setq" function. Its syntax is :

(setq <variable> <variable's_new_value>)

Registers, signals, ports and flags are all set using setq. These setq's are normally handled in parallel in a given state even though they are composed of a serial string of individual actions. This parallel action is always true in conditional action statements, and can be forced in "setq only" situations using the function PAR. Par forces parallel evaluation of all the actions and conditionals enclosed within the "par" parenthesis block. If no conditionals are contained in the state the actions are handled serially unless the par function is used. Each of these serial actions takes one complete clock cycle, therefore, in this special case a process will remain in the same named state without transition for more than one clock cycle. These successive actions can best be described as unnamed sub-states

of the named state. In MACPITTS they cannot be accessed by a direct transition due to the lack of a name. however. each consumes one clock cycle. Again Figures 2.2 through 2.5 will provide detailed examples of these action statement rules.

4. Detailed Example Explanations

MACPITTS input file code is best explained with examples. Therefore, Figures 2.2 through 2.5 have been provided as detailed examples of normal code constructs. The variables used and their definitions are as follows:

```
OFF
        (cond (time_on (go ON)))
        (cond ((= timer 5)(setq A B)(setq B A)(go OFF)))
        (cond (= A 7)(setq A 0)(go OFF)))
ON
        (cond (time_on
            (cond ((= timer max_time)(setq timer 0)(setq charge_time t)(go ON))
                  ((= A B)(go off))
                  (t (setq timer (1+ timer))(go ON)))
        (cond ((= timer 5)
            (cond ((= timer A)(setq timer 0)(setq charge_time t)(go ON))
                  (t (setq timer (1+ timer))(go ON)))
        (cond (time_on
            (cond ((= timer max_time)(setq timer 0)(setq charge_time t)(go ON))
                  (t (setq timer (1+ timer))(go ON)))
NEITHER
```

Figure 2.2 Case Conditional Evaluation and Serial Conditionals Example

time_on	An input signal (boolean)
timer	A register counter (integer)
max_time	A constant value stored in a register (integer)
charge_time	An internal signal (boolean)
A	A storage register (integer)
B	A storage register (integer)
cost	An output port (integer)
ON	A state label or name
OFF	A state label or name
NEITHER	A state label or name

Conditionals in a state are always evaluated in parallel to increase chip speed. However, in a case basis construct only the actions of the topmost conditional found to be true in the list of conditionals in a state are acted upon. All other conditionals are ignored even if their tests were also true. The signal "time_on" in the "off" state in Figure 2.2 is tested in the first conditional for a high value (true). The register "timer" is tested for a value of five and the register "A" is tested for a value of seven. If "time_on" is false and both "timer" and "A" are true only the actions following the test on "timer" are executed. It should be noted here that the equality test functions (and all test "functions") are enclosed in their own parenthesis while the non-function boolean signal "time_on" is not. All test functions must be enclosed in parenthesis.

Again, the actions following a conditional are evaluated in parallel. In the conditional where timer was tested for equality to five, the values of registers "A" and "B" would be exchanged at the end of the clock cycle because of the master-slave flip flop register design. If "A" and "B" had been signals this action construct would have been invalid because of the instantaneous activation of

signals discussed earlier.

Referring to the "ON" state of Figure 2.2 one should note that the conditionals are again evaluated in parallel. However, each leading conditional has a serial conditional construct. As mentioned earlier, in this serial construct one of the actions following the first conditional is another conditional. If "time_on" were true then the actions of this conditional would executed and all others would be ignored. At this point MACPITTS becomes confusing. A new sub-case construct has been entered due to the serial construct condition. In parallel the value of "timer" would be checked equal to the constant "max_time" and the registers "A" and "B" would be checked for equality. The final line in this block has the boolean value "t". Therefore, if all other tests in this second case basis analysis fail the action in this line will always be executed (because it is always true). If "A = B" or " timer = max_time" this always true line would never be evaluated. It should be remembered that all these second case conditionals are evaluated in parallel and all tests and actions in a state of this type must be completed in one clock cycle. The propagation delays involved in such a complex construct will reduce the maximum clock speed and the parallel constructs will increase the chip size. If "timer" had equaled "max_time" then based on the setq statements "timer" would have been loaded with zero, the signal "charge_time" would have been set true and the transition to the "ON" state would have been set. Again, the register changes and state transition will occur at the end of the clock cycle while the signal would have changed instantly.

Based on the parallel evaluation of all these actions they could have been listed in any order (say, transition first) without any difference in the results.

Two final items should be noted in the "ON" state block. The last conditional test for "time_on" at the bottom of the file would never have been evaluated because of the first case evaluation for the same test function. The last line of the file is an always true condition. If this line had not been included and all other first order conditionals had been false then MACPITTS would have employed the "fall through" method of state transition and jumped to the next state in the file (the "NEITHER" state).

a. Forced Parallel Execution

Figure 2.3 demonstrates a forced parallel execution of otherwise case basis code. The function par included under the PARALLELDEMO1 state name forces not only parallel evaluation of the conditionals, but parallel execution of the actions for every true conditional test. If "timer_on" and "A = 7" had both been true then both of their action blocks would have been executed. This parallel action only occurs for those lines falling between the opening and closing parenthesis of the par function. It should be noted that both states are equivalent in all respects even though the code appears to be different.

b. Serial Constructs

Figure 2.4 demonstrates serial constructs. These can be implemented in MACPITTS provided there are no conditionals involved in the actions. Actually, it may be possible to include conditionals in the serial code, however,

```
PARALLELDEMO1
 (par
       (cond (time_on (go ON)))
       (cond ((= timer 5)(setq A B)(setq B A)(go OFF)))
       (cond (= A 7)(setq A 0)(go OFF)))
 )

PARALLELDEMO2
 (par
       (cond (time_on (go ON))
           ((= timer 5)(setq A B)(setq B A)(go OFF))
           (= A 7)(setq A 0)(go OFF)))
 )

NEITHER
```

Figure 2.3 Forced Parallel Execution Example

serial-parallel implementation may unexpectedly occur. MACPITTS documentation in this area is very weak and further investigation may be desirable.

The flowchart system developed by this thesis currently cannot accomplish this construct as it appears. However, it can create an equivalent set of code using multiple states containing a single action each. This is covered in detail in chapters III and V. Referring to Figure 2.4, the SERIAL1 and SERIAL2

constructs are equivalent except for the state transitions. Each setq action is accomplished in 1 clock cycle. therefore. SERIAL1 will require three clock cycles before it transitions, while SERIAL2 will transition after one clock cycle. To accomplish the same actions as SERIAL1, SERIAL3 and SERIAL4 must also be completed. They each require one clock cycle, therefore, there appears to be no degradation in speed in either system. Again it should be noted that SERIAL1 uses the "fall through" state transition method to arrive at SERIAL2 on the fourth clock cycle. Also, the final results contained in registers "A" and "B" should be considered. In this serial mode at the end of the SERIAL1 state both "A" and "B" will have the original value of "B". Finally it should be realized that SERIAL5 is the equivalent of SERIAL2 because of the parallel action of a conditional.

c. Serial-Parallel

The final basic MACPITTS construct format is the serial-parallel construct. This is demonstrated in Figure 2.5. It should be noted here that not state transitions are allowed out of the conditionals. If a transition were attempted then two states would be active on the next clock cycle and that, of course. violates MACPITTS state machine rules.

5. Reset and Always Functions

The MACPITTS code provides for a specialized signal named RESET. This pin will reset all processes to their initial state (the first state listed under the

```
SERIAL1
        ((setq A B)
         (setq B A)
         (setq timer max_time))
SERIAL2
        (par
                (setq A B)
                (go SERIAL3))
SERIAL3
        (par
                (setq B A)
                (go SERIAL4))
SERIAL4
        (par
                (setq timer max_time))
SERIAL5
        (cond (t (setq A B)(go SERIAL3)))
```

Figure 2.4 Serial Evaluation Examples

```
SERIALPARALLEL1
        ((setq A B)
         (setq B A)
         (cond (time_on (setq timer A))
                (t (setq timer B)))
         (setq timer max_time))
```

Figure 2.5 Serial-Parallel Construct Example

process title line) and clear all registers, flags, and signals/ports. This reset action need not be accomplished explicitly in the action code, however, if desired it can be used explicitly as desired. This reset signal should (for all intents and purposes must) be included in all MACPITTS input files. It must be defined as a signal just as all other signals are defined with a signal name of "reset".

The "always" function is a state name and designates that this state will be executed on every clock cycle. Based on the previous rules of MACPITTS, an "always" state must be built with care, and no other state may exist in the same process.

D. MACPITTS INTERPRETER

The MACPITTS INTERPRETER can be used to verify the logic, actions and state transitions of the user generated circuit. It is activated using the "int" option when the input file is compiled using the MACPITTS compiler. Instructions for its use are included in [Ref. 2:pp. 4.30-32]. However, the reference fails to present one critical item that becomes obvious after it is presented. Prior to attempting any simulation while using the interpreter, the RESET signal must be set high (boolean true for a signal is "t") and one clock cycle must be clocked through the circuit. Then the reset signal can be set to low (f) for normal circuit operation. Failure to include the reset signal in the circuit design may or may not cause the interpreter to fail in this reset attempt. However, it becomes obvious that the constructed chip will not function correctly without this reset input

signal.

The serial action covered earlier as unnamed sub-states of a named state are indicated during the interpreter run by a + <number> following the state name. Therefore, the named state names the first action. Each sub-state action is then internally named using consecutive numbers starting with the number one.

III. FLOWCHART INTERFACE THEORY

This thesis is based on the simplicity of algorithmic state machine flowcharting methods [Ref. 6]. These flowcharts can describe in detail the available inputs. desired intermediate functions. tests, and actions, and the ensuing output results. On the other hand the MACPITTS silicon compiler was designed to allow only those fluent in a LISP style computer language to write a LISP input file describing the inputs, intermediate actions. and outputs of a state machine. MACPITTS would then convert it into a VLSI chip capable of performing this software function in hardware.

Most complex programs designed by engineers are preceded by a reasonably detailed flowchart. In the context of VLSI design this graphical method provides a good overview of the entire circuit while providing the detail required to implement it. Based on the time required to master a language with the complexity of LISP and the engineer's ready acceptance of flowcharting, it becomes obvious that. of the two methods, graphical entry of a flowchart using current CAD tools would be preferred. However, a compiler must be available that can translate the flowchart into an equivalent MACPITTS input file. Therefore. this thesis was undertaken to prove the feasibility of this flowchart-to-chip concept.

Current results have shown, within certain limitations described later in this

chapter. that any desired data manipulating system can be developed using a standard library of flowchart symbols. the SCALD graphic CAD tools currently available. and the MACPITTS flowchart compiler developed within this thesis. The following sections will describe in detail the flowchart, its symbols and the procedures required for their development and use. and limitations of the system dictated by MACPITTS.

A. SYMBOLS

The following symbols were developed using the SCALD system's body drawing capability as presented in chapter IV. Except for the TITLE and DEFINITIONS blocks. the symbols are standard forms.

1. State

MACPITTS is a state machine compiler as noted in chapter II. Each state is designated by a label or name in the user generated MACPITTS input file. Figure 3.1 shows an example of a MACPITTS input file. In this flowchart symbol library a state is designated by a rectangle and comes in three versions. The first two flip the input output pins while the third is used to designate the starting state of a process. This starting state is required any time more than one state exists within a process. A flowchart, or single MACPITTS process may have only one start state. However, it can be wired into the flowchart at any location.

Referring to Figure 3.2 one can see the available versions. Note the small square with a dot centered in it located on the left side of the symbol (highlighted

```
(program test5 3

(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def charge_time signal internal)
(def 6 power)
(def timer register)
(def time_max constant)
(def time_on signal input 5)

(process time_on_off 0

off
    (cond (time_on(setq timer 0)(go on))
          (t (go off)))

on
   (cond (time_on
       (cond ((= timer max-time)(setq timer 0)(setq charge_time t)(go on))
             (t (setq timer (1+ timer))(go on)))
        (t (setq timer 0)(go off))))

))
```

Figure 3.1 Sample MACPITTS Input File

by a small arrow). This is the state name or label attachment point. In this flowcharting system all states must have a name. This name should be unique and cannot duplicate a function or test name. This is not necessarily true in MACPITTS. The use of unnamed states as sub-states is described in chapter II. In this flowcharting system this capability is not available, however, there is no degradation because of it. All pins and attachment points of the three symbols

shown in Figure 3.2 must be either wire connected or assigned a signal name to prevent flowchart compiler errors. The puck pickup point is located at the lower left corner of the symbol. This is used when moving the symbol and is fully explained in chapter IV. The symbol name used in the SCALD is STATE or STATE.BODY.

2. Conditional

A conditional test symbol (the diamond shown in Figure 3.2) is provided to allow "if-then" tests. It has two versions that flip the "no" pin. Again, the test function attachment point is designated by a box with a dot in it (highlighted by the small arrow). Attaching a function is accomplished using the SCALD CAD graphic editor's (hereafter referred to as ged) signame option. These ged procedures are covered later in this chapter. As in the state symbol all pins must be wired into the flowchart. A disconnected symbol pin will cause unexpected and usually catastrophic results. The puck pickup point is designated by an X in the middle of the lower left symbol edge. The symbol name used in SCALD is COND or COND.BODY.

3. Intrastate

The intrastate symbol shown in Figure 3.2 is the action symbol and has two versions with the in and out pins reversed. Any action required such as setting a flag, loading a register, or activating a signal is done with this symbol. The action attachment point is the box and dot on the left hand end of the symbol (again highlighted by a small arrow). The puck pickup point is

designated by a X at the bottom left end of the symbol just before the semicircle begins to rise. The symbol name used in SCALD is INTSTATE or INTSTATE.BODY.

4. Title and Process Block

Every MACPITTS input file must include a program title and normally contains a process designator. The process designator is actually only required if subroutines are used or there are several different processes with states in each of the processes. All processes operate in parallel and are basically independent of each other. The title symbol is shown in Figure 3.3.

There are two attachment points on the left hand edge designated in the same manner as the other symbols. The top point is for the title and the lower one is for the process name. An example of the syntax and requirements of each is included in the symbol. It should be noted that the largest data path width must be included as the last entry of the title and the subroutine stack depth . if there is one, must be included in the process name. Parentheses must not be inserted in either of these entries. The puck attachment point is again the lower left corner of the symbol. There is only one version of this symbol and its name is TITLE or TITLE.BODY.

5. Signalnames

The final symbol is shown in Figure 3.3. All signals, flags, ports, registers, and the power pin must be defined in MACPITTS. This symbol is used for that purpose and must be included on any flowchart for proper compilation.

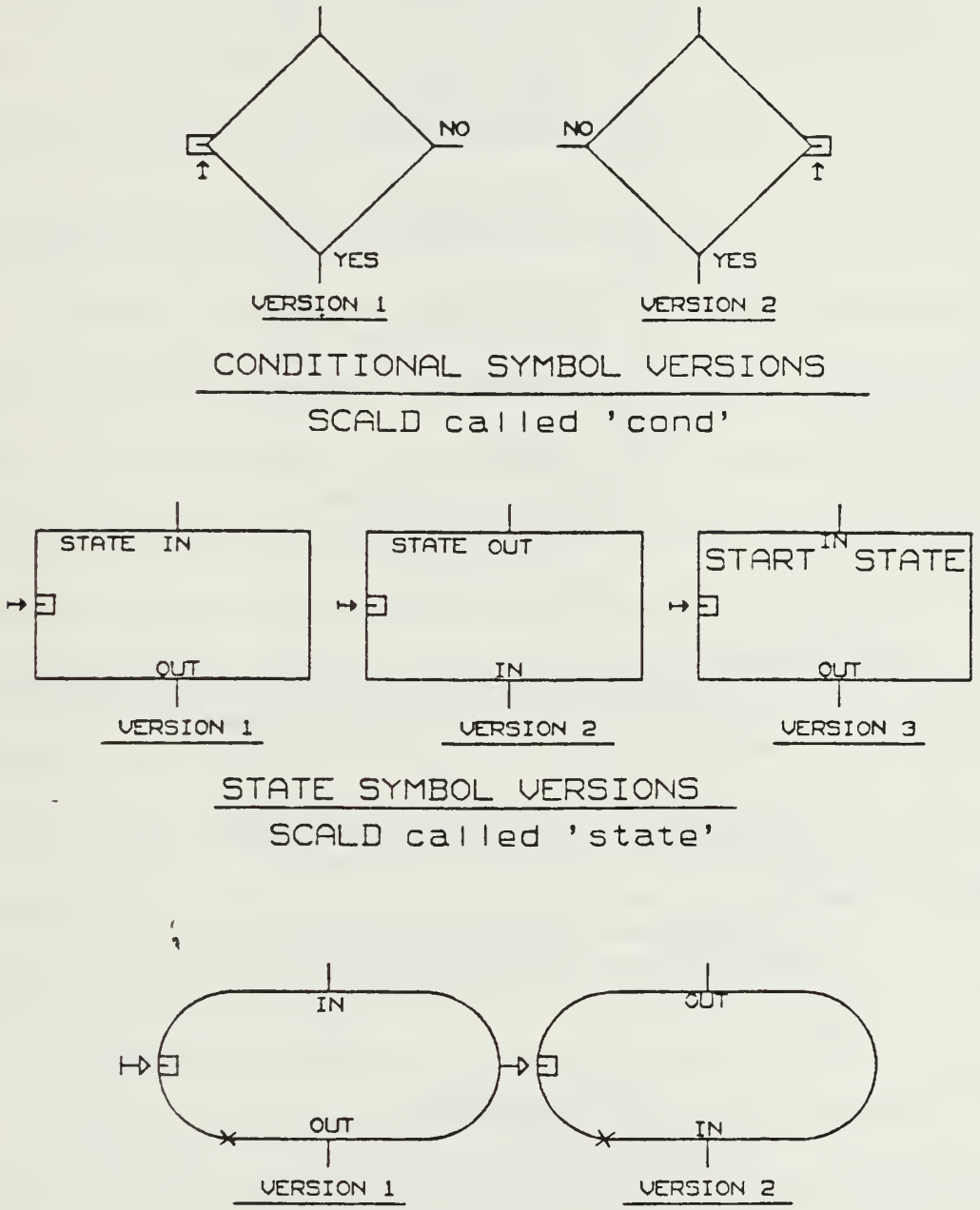
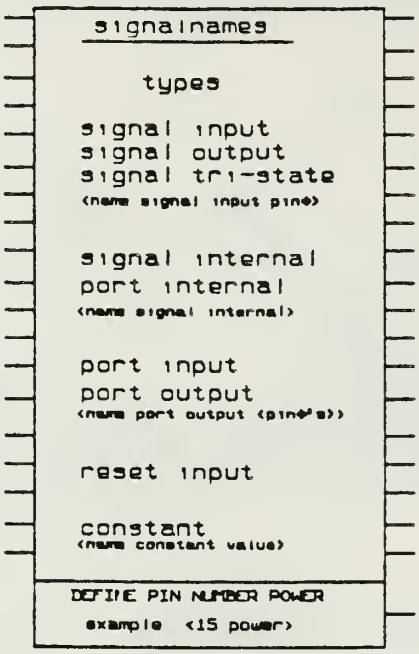
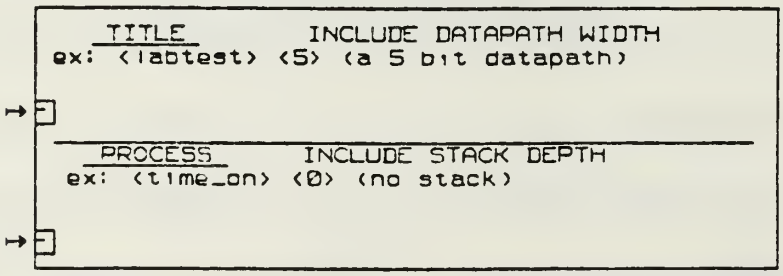


Figure 3.2 Flowchart Symbol Versions Available



SIGNALNAME DEFINITIONS SYMBOL VERSION

SCALD called 'signalnames'



TITLE SYMBOL VERSION

SCALD called 'title'

Figure 3.3 Flowchart Title and Definitions Symbols

Attachment points exist at the end of each pin and definitions can be entered in any order . one per pin. The specifics of the definitions are covered later in the chapter. The puck attachment point is located in the center of the symbol and the symbol title is SIGNALNAMES or SIGNALNAMES.BODY.

B. SPECIFIC RULES OF THIS FLOWCHART SYSTEM

The following rules were developed based on MACPITTS requirements and reduction of the overall complexity of the flowchart compiler. When the speed versus chip size tradeoffs of the MACPITTS compiler are considered. the few limitations imposed by these rules will tend to force a reasonably optimal tradeoff. It prevents extremely complex serial operations within a state (that must be accomplished in one clock period) and forces parallelism whenever possible. This increases the speed of the chip by reducing serial delay times. Also the current limitation of only one process per flowchart compiler run reduces the chip size dramatically. This limitation may be reasonably easy to overcome and suggested solutions will be included in the limitations section of this chapter.

1. Title and Process Rules

The format examples included in this symbol make completion of this requirement relatively easy. A title and the maximum data path width must be included on every drawing. The title syntax is "<program_title><maximum_data_path_width>". The data path width is designated with a decimal integer.

The process information is optional, provided there are only one process and no subroutines. If there is more than one process to be developed then the name and the user determined subroutine call maximum stack depth must be included. Each process and its states, definitions and title must be included on a new flowchart (the compiler can currently compile only one process at a time). If there are no subroutines the stack depth should be set to zero as in Figure 3.2 using the syntax "<process_name><stack_depth>". Note that only letters and numbers and the underline or dash are allowed in the block and the name must not contain any spaces. Do not insert any other text such as parentheses and periods in this block.

It should be noted that the maximum data path width must be carefully considered. It sets up the width of all registers and ports. Regardless of whether the desired size of one register is smaller than another they will all be created equal in size to the noted maximum data path width by MACPITTS. For example, if a register is going to act as the storage device in a counter it must be large enough to accommodate the highest binary number in the count.

2. MACPITTS Required Definitions

The SIGNALNAMES symbol is used to establish definitions of all the variable names used on the flowchart. [Ref. 2:pp. 11,12] provides the MACPITTS requirements. There are basically four classes of definitions, and each has a specific syntax. They are: signals, registers and flags, ports, and power definitions. All are covered in the following sections of this chapter. Some of these definitions

require chip pad numbers. Because of the ground pad and the three-phase clock pads. the lowest pad number designated on this symbol will be number five. Again. definitions can be made in any order and assigned any pad number except for ports. Ports must contain the same number of pad numbers as the maximum data path width defined in the title block and must be listed in ascending sequential order. (See Figure 3.3).

Any definition can be added to the flowchart SIGNALNAMES symbol using the ged signame option. It can be attached to any pin of the symbol. however, only one definition per pin is allow d. The ged procedures outlined in chapter IV should be followed closely.

a. Signals

A signal is a single bit boolean valued function and can be classified as one of five types; input. output. internal. tristate. and I/O. In all types except an internal signal the pad number must be designated as part of the definition. The syntax is "<signal_name> signal <type> [<pad_number >]". Here are five examples of correct signal syntax. Only alphanumeric characters and the underscore may be used with each item separated by a space. Punctuation marks or parenthesis must not be used for signal syntax.

```
time_on signal input 5
time_out signal output 6
timeoff signal tristate 7
timemachine signal internal
last_time signal I/O 8
```

The signal name RESET is a special signal. It is built into MACPITTS and

automatically initializes all processes and states when the reset pin is raised high. Therefore, states do not have to be explicitly reset, however, explicit reset may be used if desired. The reset signal should always by included on the signalnames symbol.

Signals follow some special rules in MACPITTS as noted in chapter II. Remember that signals only hold their set value during the state in which they are activated. The default setting is false and will remain so unless set to true. They are boolean set values, that is "t" for true and "f" for false, and they cannot hold a numerical value (except for an I/O type signal). This means that when a signal is set in the flowchart intrastate symbol it cannot be set to a "1" or "0", it must be set to "t" or "f" (again, except for an I/O type signal). Also, any signal defined as tristate must be explicitly set to either true or false. Unless explicitly set MACPITTS assumes the state of a tristate signal is high impedance. Finally, an I/O signal pad must be set to to be either an input or output pad by setting the signal to a 0 for output or a 1 for input using an intrastate action symbol. This selection will not change unless another explicit intrastate action changes it. This change cannot occur twice within the same state or oscillation may occur and spikes will surely be present on the output pad.

b. Ports

Port definitions are exactly like signal definitions except for two modifications. The word "signal" is of course changed to "port", and if it is not an internal port it must contain the same number of pad numbers as the width of

the data path declared in the title block. This isn't a particularly desirable trait, however, it is a MACPITTS limitation. The pad numbers must be enclosed in parentheses and be in ascending numerical order with each pad number separated by a space. The correct syntax is " <port_name> port <type> [(<pad_numbers>)]" and the following are examples.

```
time_portal port tristate (9 10 11 12)    *this is a 4 bit wide data path*
time_warp port internal
```

The parenthesis should be noted. They must be included as shown for correct syntax(asterisks indicate a comment here).

c. Registers and Flags

Registers and flags fit the same syntax. A flag is a single bit register and its size is unaffected by the data path width. As mentioned earlier all registers will be of the same size as the specified data path width. The correct syntax is "<register_name> register" or "<flag_name> flag". A flag is a boolean valued function while a register is a integer valued function. Therefore, set a flag true or false and set a register with a numerical value (be it explicit or a variable name).

d. Constants and Power

To provide the maximum separation between the ground pad and the power pad on the chip the ground pad is always designated as pad number 1 (automatically by the compiler). The power pad is designated by the user as the highest pad number on the definitions block. The power syntax is

"<highest_design_pad_number> power". This is opposite all other signal syntax. The pad number comes first for the power pad.

A constant may be required in the system and is defined using the following syntax "<constant_name> constant <constant_value>". An example:

stop_time constant 60

This constant is actually a hardwired register and as such must meet the maximum data path width requirements mentioned earlier. In this case the data path must be at least six bits wide to obtain a binary representation of 60 decimal.

C. STATES

A state is basically the state of the system as designated by the position of the state in the flowchart. Actions take place only when the state in which they are contained is active.

1. Flowchart Implementation

In this flowchart system any state is designated by a name chosen by the user and placed in the state symbol. It is attached to the symbol at the designated attachment point as a signal name using the GED SIGNAME function. Use of names longer than one letter will reduce user errors and the following restricted names must not be used by themselves:

setq	process
in	title

out	outstart
cond	outother
a	yes .
no	any number

This limitation is required to reduce the chance of error in the flowchart compiler. These names may indeed function without difficulty, however, an exhaustive test of the compiler using such names has not been completed.

As discussed previously the third version of this symbol is titled START STATE and must be used to designate the startup state to prevent problems when the states are initialized using the reset signal. Without it the choice of the starting state will be made randomly. skewed toward the last state entered during flowchart development.

2. State Content

The operations/actions included under a given state are dictated by the MACPITTS compiler requirement. A thorough understanding of MACPITTS is not required. however. the following rules will make more sense if chapter II is fully understood.

Directly under a state symbol two different symbols may be wired, a conditional and/or an intrastate. Each of these will be discussed in later sections. There is no theoretical limit to the number of parallel intrastates and/or conditionals that may be placed within a state. However, chip size is influenced by this parallel construct and too many may create a MACPITTS chip that cannot be manufactured using current vlsi technology. Figure 3.4 provides a

complex example of this parallel construct. Only one level of intrastate action symbols may be implemented. i.e., serial intrastates are not authorized. Two levels of conditionals are currently legal and in essence provide a dual case basis algorithm. It is termed dual because the second conditional can be attached to either the yes or no terminal of the first conditional. A current limitation which is documented in the FINALSORT directory of the compiler code in Appendix B prevents more than two conditionals in a case or series construct. This can be corrected as suggested in the same code documentation and in chapter V. Figure 3.5 shows two illegal constructs. It should be noted that Figures 3.4 and 3.5 contain no function or action statements. They are presented to show acceptable and unacceptable layout geometries.

3. Symbol Differences

All symbol versions (except the START STATE version) in this flowchart system are minor revisions of the symbol pinout placements. The start state symbol, however, has special meaning and one must be included on every flowchart. It designates the initial state MACPITTS enters when the chip is enabled. All actions (intrastate symbol) must transition to a state. Therefore, only the starting state is critical because all other state transitions are driven by the action code. This forced state transition is much like the traditional "goto" statement. Chapter I explains the MACPITTS input file with specific examples that reinforce these limitations.

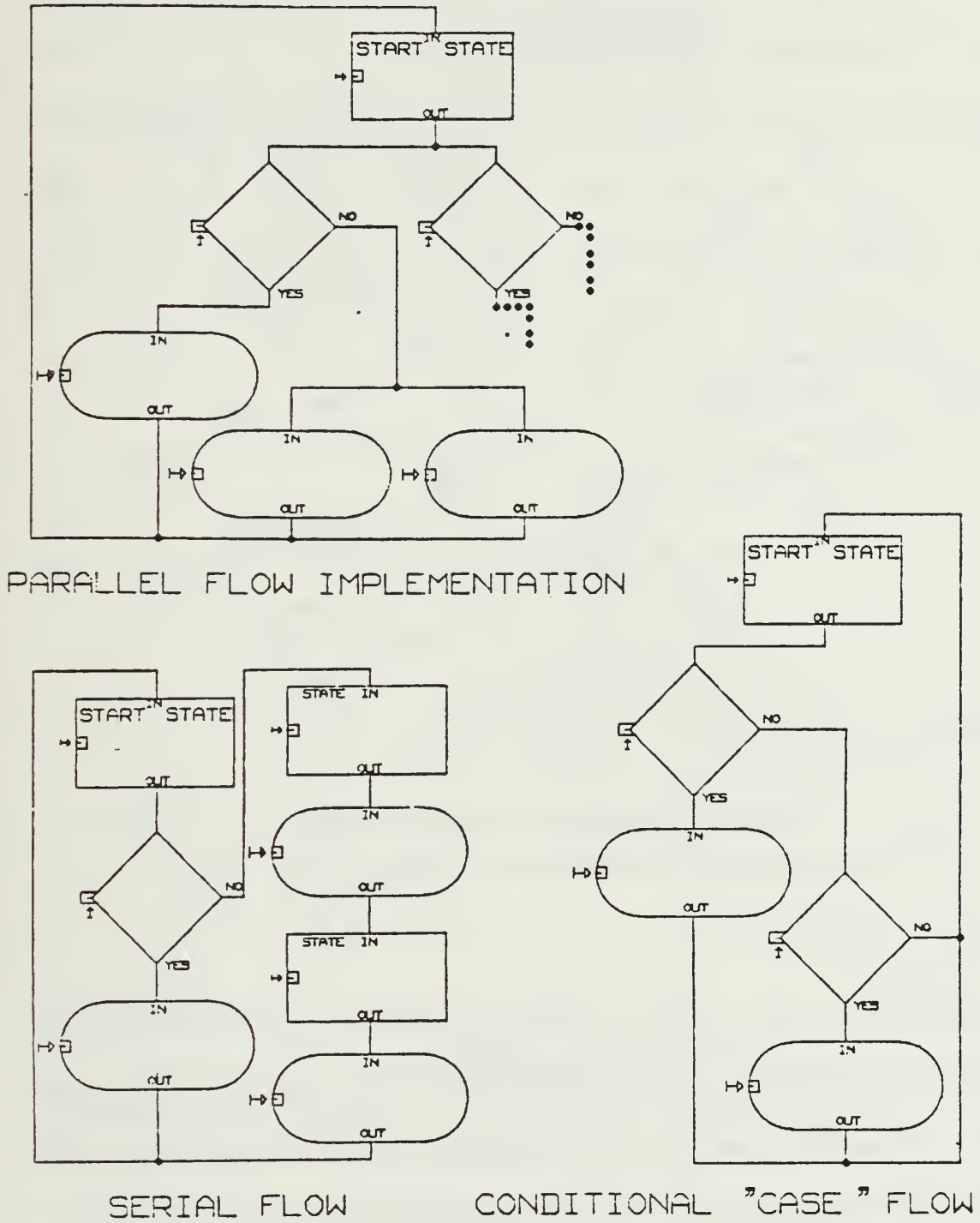


Figure 3.4 Acceptable Parallel, Serial and Case Constructs

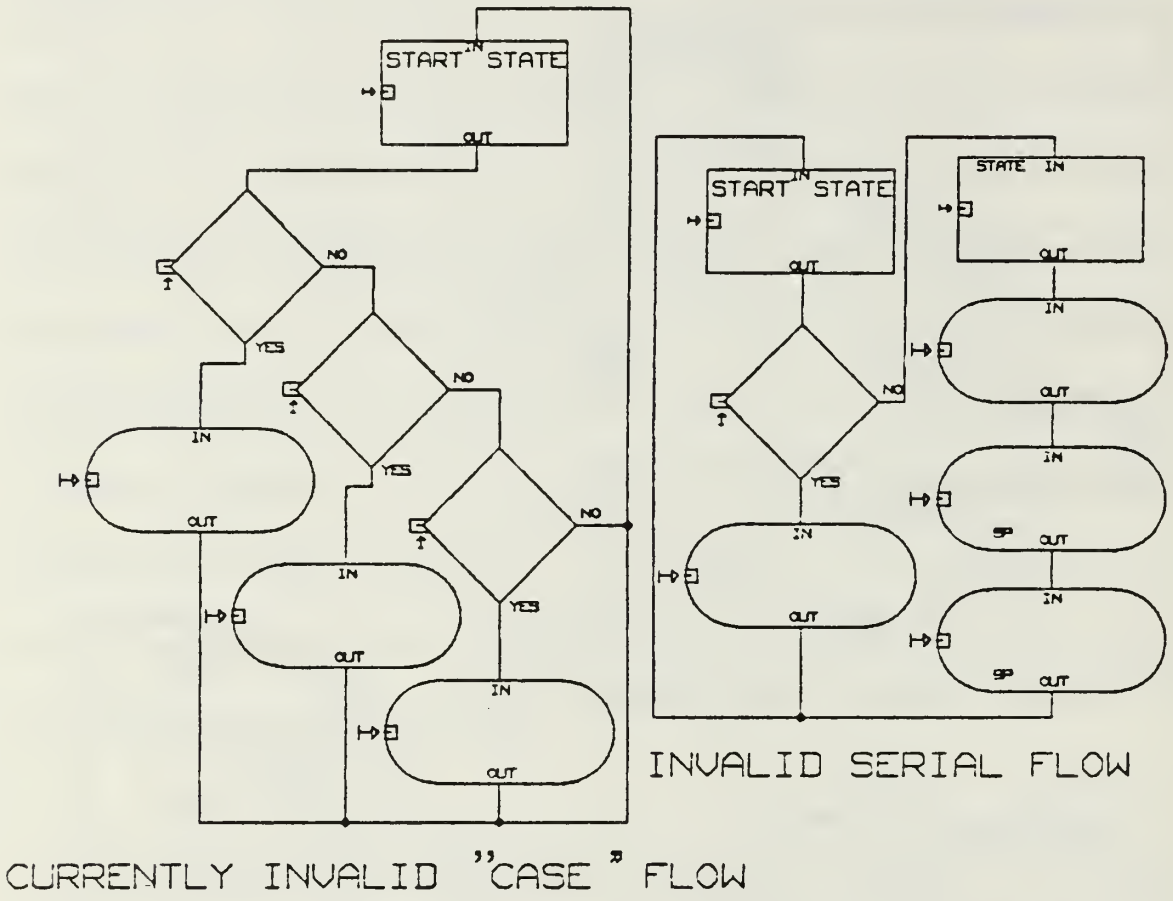


Figure 3.5 Unacceptable Flowchart constructs

D. Conditionals

Conditionals can be placed in series or parallel with some limitations as mentioned above. The case basis construct must follow Figure 3.6. Any successor must be connected to the "no" pin of its ancestor. In standard algorithmic code it is an "else if" construct:

if a is true then (action, transition to next state)

else if b is true then (action, transition to next state)
else if c is true then (action, transition to next state)
else (action, transition to next state)

MACPITTS evaluates all these tests in parallel and executes the first true statement (topdown evaluation). Currently only two cases can be evaluated within a state. This is a flowchart compiler limitation and not due to MACPITTS. Suggested solutions which require additional routines in the

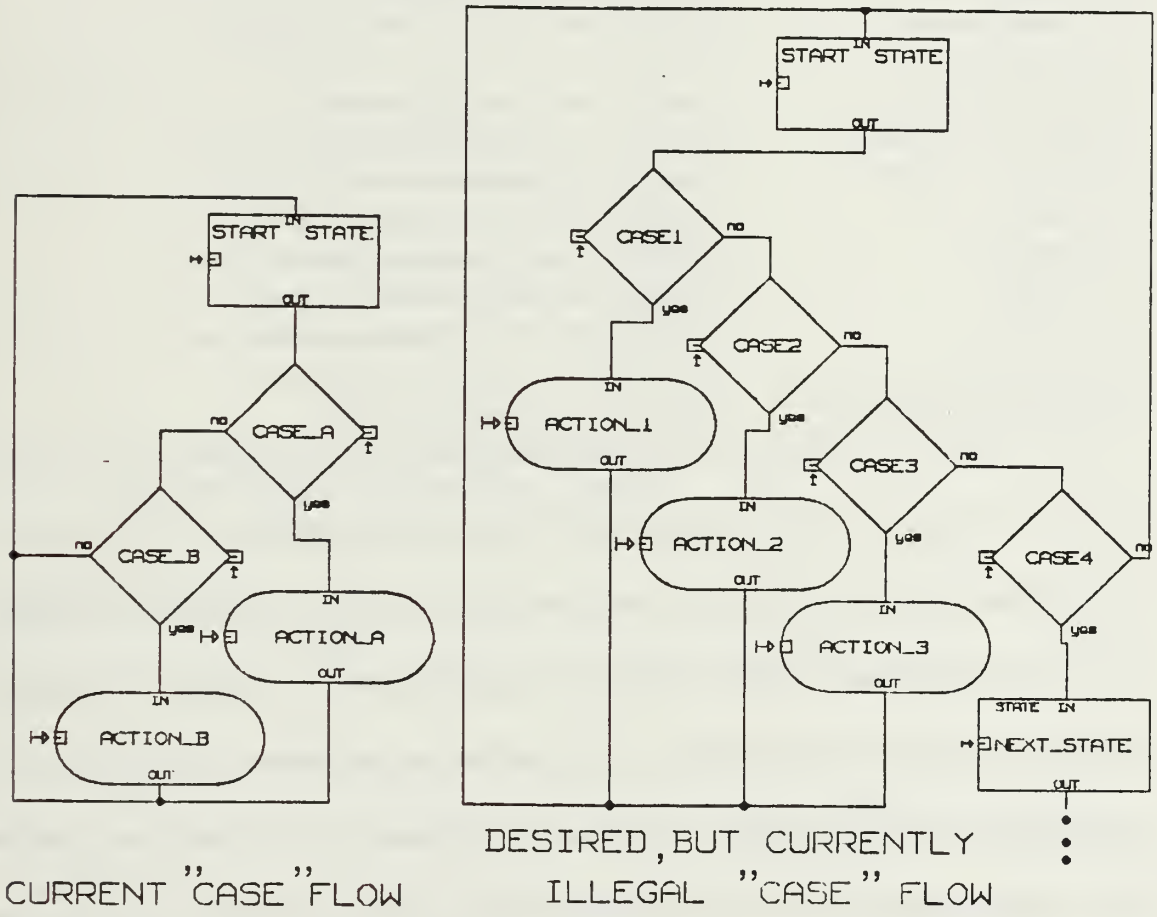


Figure 3.6 Example of Currently Acceptable and Desired Case Constructs

compiler are presented in chapter V.

1. Conditional Test Capabilities

Any of the following tests [Ref. 2:pp. 47-50] can be made subject to the proper syntax:

SYNTAX	MEANING
boolean variable	If the boolean value is true return a boolean t
(= integer integer)	If both integers are = return a boolean t
(=0 integer)	If integer = 0 return a boolean t
(<> integer integer)	If both integers are not = return a boolean t
(<>0 integer)	If integer is not = 0 return a boolean t
(unsigned-< int1 int2)	If int1 is < int2 return a boolean t
(unsigned-<= int1 int2)	If int1 is < or = int2 return a boolean t
(unsigned-> int1 int2)	If int1 is > int2 return a boolean t
(<0 integer)	If integer is < 0 return a boolean t
(<=0 integer)	If integer is < or = 0 return a boolean t
(>0 integer)	If integer > 0 return a boolean t
(>= int1 int2)	If integer1 is > or = int2 return a boolean t
(>=0 integer)	If integer is > or = 0 return a boolean t
(eq int constant (bit-pos-list))	If int is = to the constant in the bit positions listed in the bit-pos-list return a boolean t. If the constant is an ascii character within " " it's ascii number value is used.

All of these returned results are sent to the MACPITTS controller. Therefore, these test statements will not modify the data path results. An action (intrastate) statement is required to modify any data path element content. Any function or variable can be substituted for the integer or boolean value in these tests provided they are of matching type. The possible functions are listed in the intrastate section of this chapter. Extremely complex tests can be made based on the user's desires and the correct syntax. The syntax of the entry in the conditional symbol must follow the test syntax presented above. This specific syntax requirement is a

current drawback of the flowchart compiler. Future work in this area would allow standard algorithmic entries and alleviate the somewhat confusing syntax. See chapter V for suggested improvement methods and ideas.

Examples of conditional tests are:

```
(= a b)                  a and b are integer variables (possibly registers)
time_on                    time_on is an input or internal (boolean) signal.
                             This asks if time_on is a boolean true.
(<>(8<< 0 counter))   Shift the counter integer 8 bits left filling the
                      LSB's with 0 and test this result for the
                      condition not = 0.

(eq(word-nand counter portinput) 61 (5 3 1))
                       This says take the bit
                      wise nand of the 2 integer variables counter
                      and portinput (possibly a register and a port
                      input) and test the binary bit positions 5 3
                      and 1 for equality between the nand result
                      and the binary equivalent of the decimal
                      constant number 61.
```

Again. the complexity of the conditional test is limited only by the user's desires. It should be noted that numerous operations were done in the last example above. It still is not classified as an action (intrastate) statement because the result is a boolean true or false returned only to the controller section of the chip.

The last conditional in the list above indicates that an ascii character enclosed in quotes will produce the ascii equivalent number. The SCALD system uses quotes as field separators in the connectivity file it produces when the flowchart drawing is written to a directory. To prevent compiler confusion all quotes are forbidden by SCALD and are actually ignored if they are entered. Therefore, this last option is unavailable at this time. Please refer to [Ref. 2:pp.

47-50] for a full explanation of these functions.

2. Symbol Configuration

All test functions discussed in the last section are attached to the special attachment point designated by the small arrow near the conditional symbol. All pins must be either wired into the diagram or remotely connected using a common signal name for the pins connected together. Also the SCALD system may not allow the leading parenthesis when typing the function. If an error message is generated, then enter the function without the offending parenthesis and use the ged change command as described in chapter IV to add the parenthesis. The state section and Figures 3.4, 3.5 and 3.6 cover the possible layout geometries for conditionals.

E. INTRASTATE

Intrastates are the action symbols of this flowchart system. Any required action is accomplished within a state using this symbol. Registers are loaded or changed, signals ports and flags are set and I/O pads are direction selected. They can be assembled in parallel, but not in series unless a state separates them. Figures 3.4 and 3.5 demonstrate this. MACPITTS actually allows multiple series actions within one state if no conditionals are included, however, there appears to be no speed or hardware advantage to this so this flowchart limitation will not be considered a limitation. Also this parallel only action criteria corresponds to normal state flow concepts.

Chapter I covers the differences among signals, ports, flags, registers and constants. While not essential, a reasonable understanding of these differences will prevent problems when working with this flowchart system. [Ref. 3] also addresses the capabilities and limitations of MACPITTS and may provide more insight.

The following actions are available:

ASSIGNMENT ACTIONS

variable1 variable2	set variable1 = to variable2
boolean_variable t	set the boolean variable = to a boolean true
integer_variable 1	set the integer variable = to 1 (as in a flag)

ARITHMETIC ACTIONS

(1+ integer)	increment the integer (a register...)
(1- integer)	decrement the integer
(+ integer integer)	add the 2 integers
(- int1 int2)	subtact int1 from int2

SHIFT ACTIONS

(<< boolean integer)	left shift 1 bit fill with boolean value
(2<< boolean integer)	left shift 2 bits fill with boolean value
(3<< boolean integer)	shifts 3,4 and 8 are available
(>> boolean integer)	right shift 1 bit fill with boolean value
(2>> boolean integer)	shifts 2,3,4 and 8 are available

LOGICAL ACTIONS

(nor boolean ... ...)	logical nor of 2 or more boolean variables
(nand boolean ... ...)	logical nand of 2 or more boolean variables
(or boolean ... ...)	logical or of 2 or more boolean variables
(and boolean ... ...)	logical and of 2 or more boolean variables
(equ boolean ... ...)	true if all 1 or all 0
(xor boolean ... ...)	the xor of boolean variables
(not boolean)	the inverse of the variable, note only one
(parity boolean .. ..)	returns 1 for odd and 0 for even parity
(word-nor int int)	the bitwise nor of 2 integer variables
(word-nand int int)	the bitwise nand of the 2 integers. All

of the boolean functions above are available
for integers except for the parity.

All of the above functions are covered in [Ref. 2:pp. 5-50] and [Ref. 4]. [Ref. 2:pp. 47-50] contains some errors. The functions are correct in the list above. Again, the parenthesis must be included when these functions are used on the flowchart. Complex actions can be accomplished in the same manner as the example of the complex tests discussed earlier. The action or test function doesn't have to fit within the symbol on the flowchart as long as it is attached to the correct attachment point.

IV. SCALD SYSTEM

The SCALD graphics system is a complex and versatile CAD tool. It was used as the MACPITTS Flowchart development device and contains all the required diagram symbols in a locally developed library. While Appendix A contains detailed information about the system and its use, the following paragraphs provide a quick reference description of the features needed for MACPITTS Flowchart development.

A. SCALD CAPABILITY OVERVIEW

SCALD can be used to design circuits using standard MSI or LSI components. These designs can then be tested for both logic and timing using the logic simulation and timing verifier routines. The timing verifier requires a more complex, user provided, input file which should include items such as wire delay estimates, setup and hold time estimates, initial signal levels and the clock period desired. This complexity allows more flexibility in the designer's test procedures, but also requires a deeper understanding of the circuit and its components and normally more user effort. After the circuit has been tested to the designer's satisfaction SCALD has a packager routine which provides a number of output files designed to make actual circuit package wiring easier. It provides a listing of the external pin connections by pin and chip number based on the SCALD's best

estimate of minimum package use. The packager routine assumes that no changes in the circuit design will be made and locks all generated output files. These files can not be overwritten. therefore. if a second packager run is desired using the same file name. the old output files must be unlocked and deleted before the second run is conducted.

SCALD includes extensive MSI or LSI libraries for each of the capabilities mentioned. however. it has only one compiler routine. If the compiler is run to check the circuit logic and then run for circuit simulation the logic results in the compiler output files are overwritten with the simulation files. These files are the CMPLOG.DAT.CMPEXP.DAT CMPLST.DAT and CMPSYM.DAT. Refer to Figure 4.1 for a flow diagram of the complete system.

Applying these libraries and capabilities to this thesis was of very limited use. There was neither a library of flowchart symbols nor a corresponding simulation library. Using the system's body construction capabilities (this capability will be covered later), a local library of flowchart symbols (or bodies) was developed and include:

conditionals	2 versions
states	3 versions
intrastates	2 versions
signal name definitions	
title	

Any flowchart drawing for MACPITTS must include at least the last four symbols for proper conversion to a MACPITTS input file. The MACPITTS simulation capability (provided by the MACPITTS interpreter) is considered

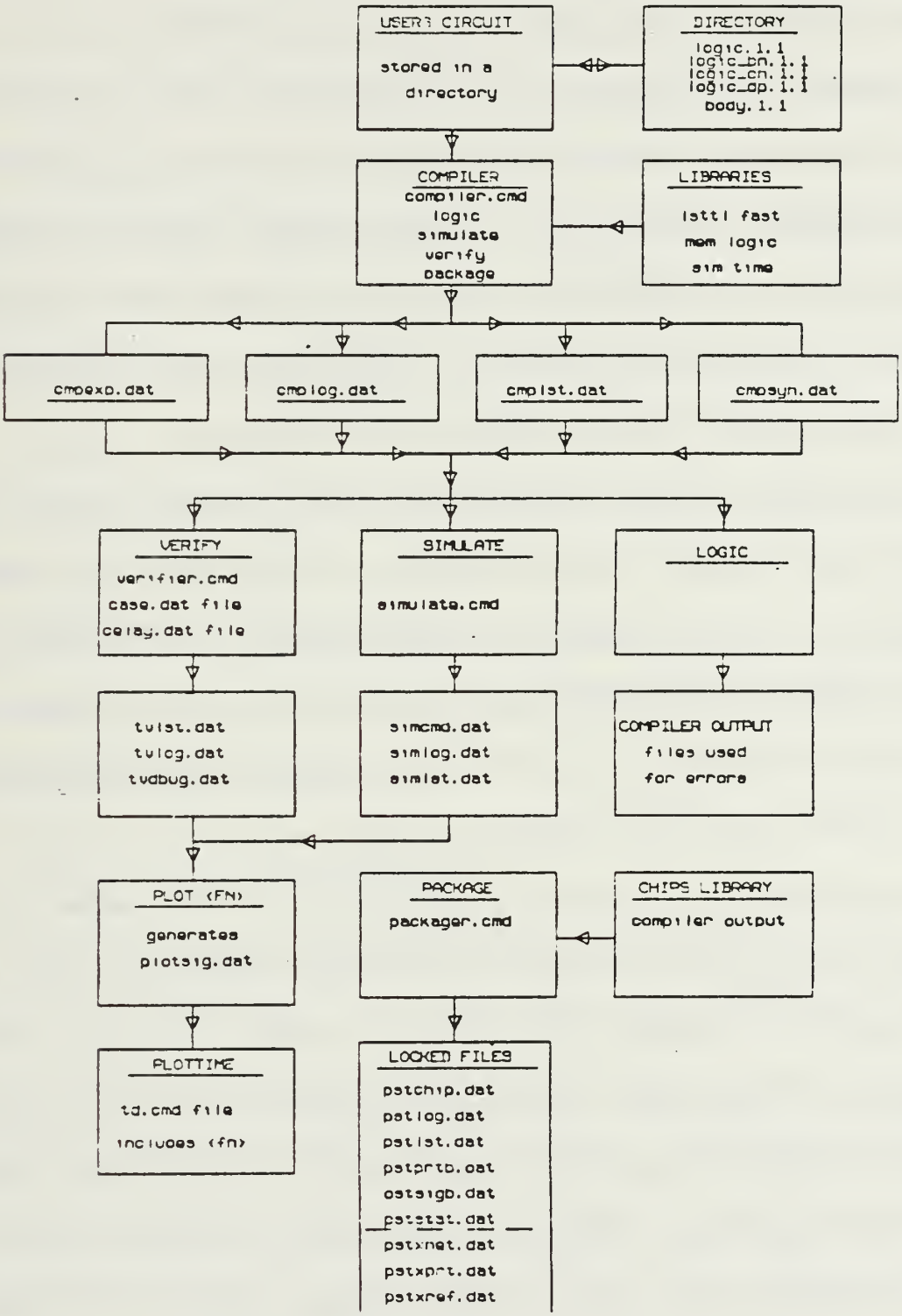


Figure 4.1 Scald System Overview [Ref. 1]

adequate for required simulations. therefore. a simulation capability and its requisite library have not been developed for the SCALD system.

Throughout this document the term SYMBOL and BODY are considered synonymous in that a flowchart symbol is constructed using the SCALD system's body capabilities. Each body has two distinct features other than its graphic shape. They are its PROPERTY and any number of desired input/output PIN NAMES. When creating or reviewing a body these features can be seen using the ged commands SHOW BOTH and SHOW. Refer to Figure 4.2.

In this locally created symbol library the property feature was not used other than to give the body a title in the library. All other required and desired functions were developed using the pin names capability. When the bodies are used to create a MACPITTS flowchart on a logic drawing (<file_name>.logic) the properties and pin names are attached to the symbol. but are suppressed from view. When a signal name is attached to a symbol pin on the logic drawing it is automatically tied to the corresponding pin name by a SCALD assigned signal number. The MACPITTS flowchart and subsequently the flowchart compiler use this capability by requiring any MACPITTS function or component to be designated as a signal name. This may sound confusing because, for instance, a register or conditional function is not a signal, however, using the SCALD system the SCALD signal name or SIGNAME function is used to implement these non-signal assignments. The specific symbol determines the type of operation performed on these signal named functions such as the MACPITTS cond and setq

functions. Refer to Figures 4.2, 4.3, 4.4 which show a very short and incomplete flowchart drawing, the corresponding body, drawing and the connectivity file produced by the SCALD system when the logic drawing is written to a directory.

The connectivity file is used as the input file for the flowchart to the MACPITTS compiler. One should note the rather strange signal names listed in Figure 4.4. These are generated by SCALD when wires are used to connect two symbol pins together (in this case no signal name is given by the user). When a logic drawing (synonymous with a flowchart drawing) is written or saved to a file, a distinct property number is attached to each symbol on the drawing. This number (easily distinguished as the number with the P behind it, as in 24P) is used along with the signal names, corresponding signal numbers, and pin names by the flowchart compiler to create the MACPITTS input file. Examples of each symbol are included in Appendix B.

B. THE SCALD GRAPHICS EDITOR

GED is the SCALD graphics editor mode and is used when creating any circuit or flowchart drawing. When called using "ged" under the unix C Shell prompt (%csh) a graphics page with grid lines and options appears. Two types of drawings .BODY and LOGIC, may now be drawn. Those terms will appear in the title of the drawing as dot extensions. An example of a logic drawing title is; FLOWCHART.LOGIC.1.1, the first version of a logic drawing named flowchart. The following sections provide detailed procedures for ged operations.

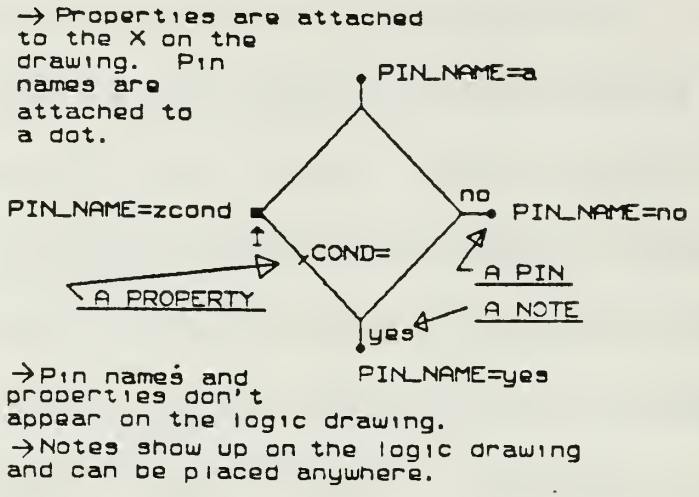

→ Properties are attached to the X on the drawing. Pin names are attached to a dot.
PIN_NAME=a
PIN_NAME=zcond
no
PIN_NAME=no
COND=
A PIN
A PROPERTY
A NOTE
yes
PIN_NAME=yes
→ Pin names and properties don't appear on the logic drawing.
→ Notes show up on the logic drawing and can be placed anywhere.

Figure 4.2 Example of a Body Drawing or Flowchart Symbol

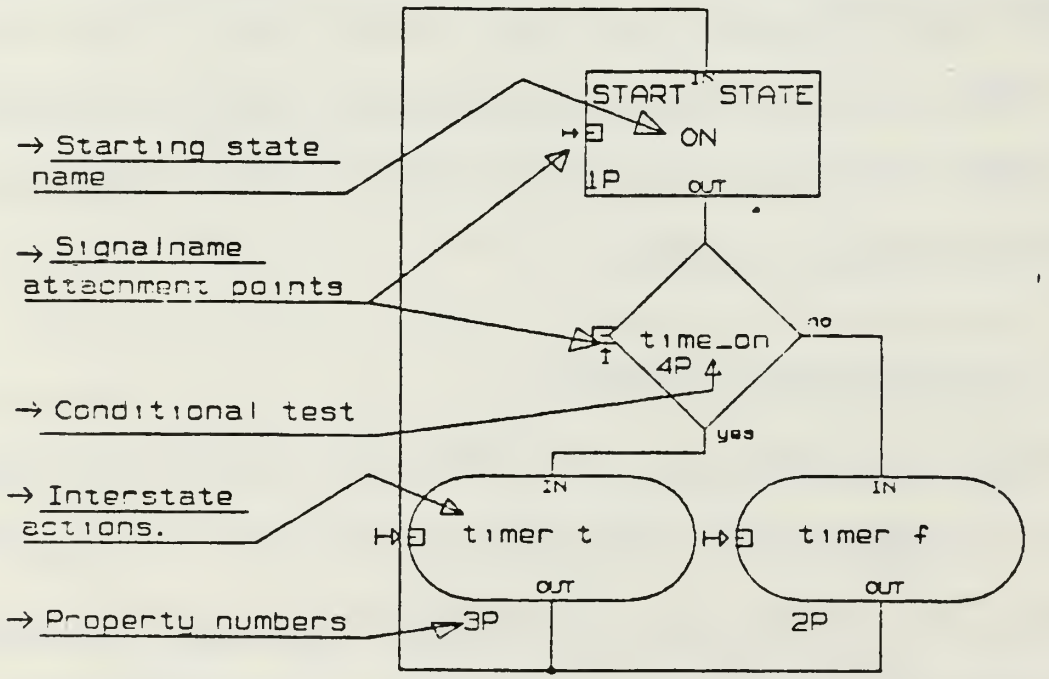


Figure 4.3 Example of a Logic or Flowchart Drawing

```
FILE_TYPE = CONNECTIVITY;
(GED release: 7.27TFinal:Tue Jan 14 19:48:22 PST 1986)
Ø"NC";
1"time_on";
2"timer t";
3"timer f";
4"ON";
5"UN$1$COND$4P$no";
6"UN$1$COND$4P$yes";
7"UN$1$COND$4P$a";
8"UN$1$INTSTATE$2P$out";
%"STATE"
"3","(2ØØØ,325Ø)","Ø","weist.wrk","1P";
;
STATE" ";
"zgoto"4;
"outstart"7;
"in"8;
%"INTSTATE"
"1","(26ØØ,195Ø)","Ø","weist.wrk","2P";
;
INTSTATE" ";
"setq"3;
"out"8;
"in"5;
%"INTSTATE"
"1","(1675,195Ø)","Ø","weist.wrk","3P";
;
INTSTATE" ";
"setq"2;
"out"8;
"in"6;
%"COND"
"1","(2225,265Ø)","Ø","weist.wrk","4P";
;
COND" ";
"zcond"1;
"yes"6;
"no"5;
"a"7;
END.
```

SIGNAL NAMES BY NUMBER.

THE PROPERTY NUMBER.

THE PROPERTY NAME.

THE PIN NAMES AND SIGNAL NUMBERS.

Figure 4.4 Example of a Connectivity File Created by Figure 4.3

1. Screen Functions

All routinely used screen editing functions appear as blocks along the right hand border of the screen. These include such items as version, split, copy, window, property name, signame (signal name), change, and a final block at the bottom that the user can change as needed. Examples of items that appear in this lower block are add, note, set, write, and hardcopy.

Again. [Ref. 1:pp. 3.3-4.79] and Appendix A contain amplifying information.

2. Puck or Mouse Use

The puck .or mouse. has 4 buttons used for function selections and its position on the table determines the cursor location on the screen. Since the cursor position is based on an electric field generated by a wire grid located under the table and received by the puck, a position on the screen will always correspond to a specific point on the table. Lifting the puck in effect kills the cursor until the puck is again placed on the table over the grid area.

All lines, circles, wires, symbol shapes etc. are drawn using the puck and all text (both active, as in signal names, and passive, as in notes) is placed using the cursor driven by the puck. The four buttons perform the functions presented below.

a. Yellow

The yellow button starts and stops a line or wire at a grid intersection whether or not the grid is visible. All wires and lines are automatically drawn on a grid line. This button is also used when moving an object. wire, or symbol because it allows these items to be oriented on the grid line system facilitating wire lineup.

b. Blue

The blue button starts and stops lines or wires at a vertex which is the end of a body pin or a wire. Using this button guarantees that the line is connected to a wire or pin and not at a grid intersection close to them. User

required cursor accuracy is reduced because the end of the wire being drawn will snap to the vertex closest to the cursor. This cursor proximity property can be used to great advantage when placing intersection dots.

c. Green

The green button changes the wire bend direction. The three possibilities are an upper corner, lower corner, or straight line between two points. Pressing the button cycles through the bend versions.

d. White

The white button is used for group moves. Assuming that a group has been previously designated, using the GROUP option and the yellow and blue buttons, it can be moved using the MOVE option and the white button. The cursor must be located within the group boundary before depressing the button. The time required for the system to redraw the moving group depends upon the number of elements in the group.

3. Logic Versus Body Drawings

A logic drawing is just what it implies, a logical interconnection of a set of symbols representing some logical function. These interconnections are accomplished using named pins on the symbols and "wires" or lines to tie these pins together. Test signalnames can be attached to a pin in lieu of a wire if desired. The MACPITTS flowchart is a logic drawing created using symbols previously created in body drawings and stored in a local library of symbols. A body drawing, therefore, is in essence a symbol used on logic drawings. However,

body drawings cannot be used to create other body drawings without use of the SMASH option. A more detailed description is presented later in this chapter.

A symbol in the flowchart library contains the symbol property name, signal pin names, and any notes that will be presented with the symbol when it appears on a logic drawing. Pin names are not visible on the logic drawing. Also the property name can be suppressed, if desired to make the symbol less cluttered, by using the SPLIT option and deleting the name.

A logic drawing can be checked for basic pin interconnect errors by using the CHECK and ERROR keys on the right hand keypad of the keyboard. If an error appears it must be corrected. Any error will create havoc during the flowchart compilation into MACPITTS. On the left hand keypad there are three keys, the SHOW, SHOW ATTACH, and SHOW BOTH which can aid in checking signal name attachment points (and pin names on body drawings) if desired. Finally, items such as property numbers can be suppressed prior to writing (saving) a drawing by designating the area you want suppressed as a group and using the command DISPLAY INVISIBLE. This is useful when preparing presentation quality graphics. Refer to [Ref. 1:pp. 3.3-4.79] and Appendix A.

4. Creating New Bodies and Versions of Bodies

There are two basic starting points when creating a new symbol using the body creation capabilities of SCALD. The first is based on creating a new symbol (usually an MSI or LSI chip) based on a complex multi-gate logic diagram

previously created by the user. The designer is creating a single symbol with input/output pins that correspond to the input/output signals of the larger more complex logic drawing. This completed symbol can then be used in place of the larger logic drawing on any logic drawing desired. Detailed instructions on this method are available in Appendix A and [Ref. 1:pp. 3.3-4.79].

The second starting point is based on nothing more than a desire for a symbol with a specific input/output and an internal property. It can serve any desired purpose provided the goals meet the capabilities of the symbol.

The symbols created for the MACPITTS flowchart system were developed based on this second form of the body drawing. It must be remembered that these symbols can only be used on logic drawings.

a. The Body

A body can be edited or created by entering "edit <bodyname>.body" while in ged. The < >'s mean that any file name may be used between the < >'s and they are not included in the command.

An X and the <bodyname> will appear in the center of the screen. The X designates the origin of the symbol. If the property name (the <bodyname>) is not desired in the logic drawing presentation of the symbol then the SPLIT command can be used, and after moving the cursor over the X, the blue button should be depressed twice. The <bodyname> can now be moved away from the X and deleted using the DELETE option. The X should not be moved. Movement will create more problems then it can solve. If one desires to

move the origin after the symbol is completed it is easier to start over. Shifting the origin will also cause problems with the creation of other versions of the same symbol.

The grid size listed at the top of the window must be checked. It must be equal to the grid size of the logic drawings to prevent wire and pin matchup problems on the logic drawing. The default size of the body drawing is 0.024 1 while the default size of logic drawing is 0.05 1. The grid size of the body can be changed using the command GRID SIZE 0.05 1 before adding anything to the body drawing.

Use of the WIRE option and the puck allows the user to draw the symbol outline. Connection point pins can be added as needed if one draws a short wire at any desired location on the periphery of the symbol and attaches a dot to the end of each pin. If the signal name (SIGNAME) function will be used for purposes other than just designating input and output signals then some type of annotation symbol must be developed to denote to the user the difference between these "function" pins and normal input/output signal pins. This capability is used on all the MACPITTS flowchart symbols except the signalnames (definitions) symbol. This specialized attachment point is currently denoted by a small box drawn around a short pin with the end dot located in the center of the box and is highlighted by a small arrow.

Pin names can be attached to each pin through the use of the SIGNAME option. Before text is entered from the keyboard and the return key is

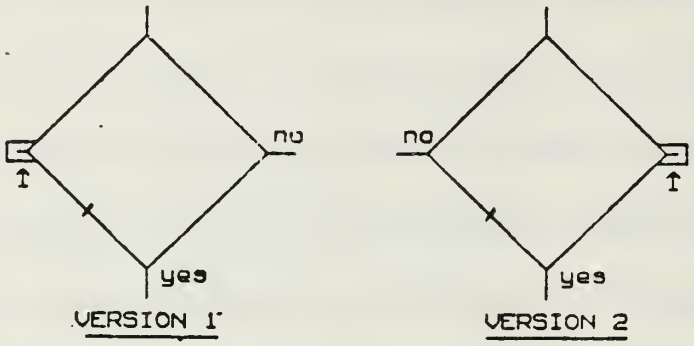
depressed. the pin dot should be selected with the cursor and the yellow button. This will guarantee that the pin name is attached to the correct pin. It should be noted here that these pin names appear on the connectivity file (logic_cn.1.1 file) in reverse alphabetical order grouped by property number. Refer to Figure 4.4. If a specific order is required in the connectivity file for use in the compiler, then serious consideration should be given to the selection of these pin names. They can be changed at a later date with extreme caution. The current use of zcond and zgoto in the conditional and state symbols is based on this reverse order listing phenomena.

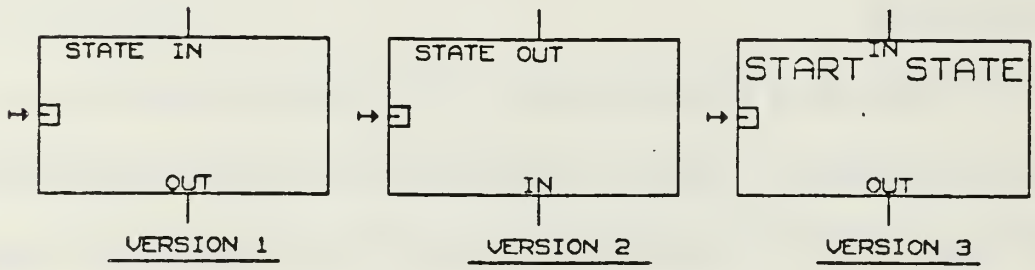
Notes should be added sparingly to the symbol to make it easier to use on a logic drawing. It should be remembered that the notes and undeleted property names show up on the logic drawing while the pin names do not. When the body drawing is complete it can be saved by issuing the command WRITE.

b. Versions

Creating another version of the same body allows one to use the VERSION option during logic drawing development. Use of the cursor and the yellow puck button will cause the available versions to appear in place of the currently displayed symbol. Figures 4.5 and 4.6 present the current versions of the MACPITTS flowchart symbols.

A version is created by editing a "<bodyname>.body.*.1" where the * is replaced by the version number. Although the method of adding a previously designed body to the new version and then using the SMASH command to





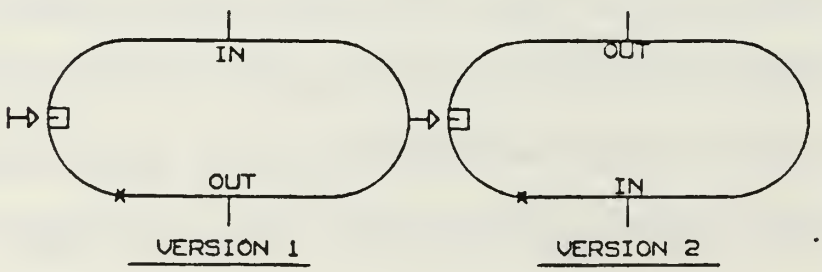
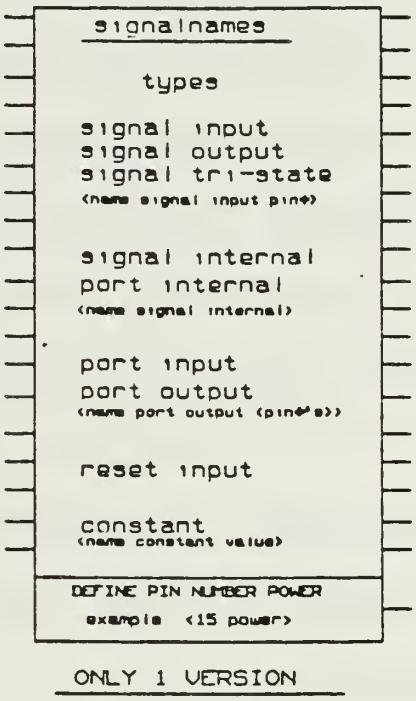
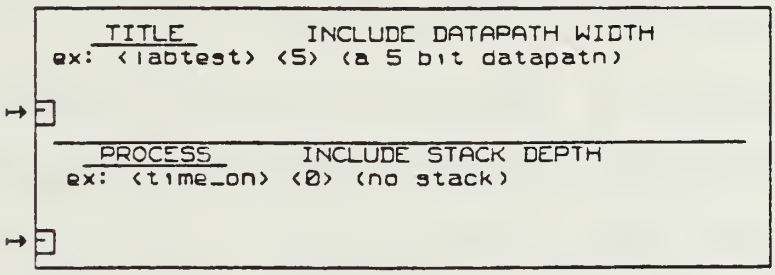


Figure 4.5 Conditional, State and Intrastate Symbol Versions



SIGNALNAME DEFINITIONS SYMBOL VERSION

SCALD called 'signalnames'



TITLE SYMBOL VERSION

SCALD called 'title'

Figure 4.6 Signal Name Definitions and Title Block Symbol Versions

remove all the added symbol's pin names and property is acceptable. there is a better method available for creating versions. This is accomplished by editing the original body version and modifying the write command to include the new version name after write. An example; write <bodyname>.body.2.1. This creates a copy of the original which can now be modified as desired using the standard ged commands available. The use of SHOW ATTACH is prudent after the changes have been completed to ensure that all new pin names are attached as expected. The only way to effectively change or move a pin and its pin name is to delete them and redraw as necessary. This procedure guarantees that the basic symbol and its origin don't change from version to version. The origin of every version must be in the same location.

5. Creating Logic Drawings

Creating a logic drawing is relatively simple. While in ged issue the command "edit <logic_drawing_name> and a blank page or previously drawn logic drawing with that title will appear ready for editing. Use of the ADD <symbol_name> command will add the <symbol_name>.body symbol to the logic drawing. Several copies of the same symbol can be added by depressing the yellow button once to produce a moving copy and pressing it again to place it. Drawing speed is increased if several copies of the same symbol are placed on the drawing near their expected final position before the next symbol type is added. The version option can be selected to view the available versions one at a time as the yellow puck button is pressed. This presumes that the cursor is within the

chosen symbol. These symbols can be moved easily using the MOVE option. The body origin (the X in the body drawing of the symbol) is where the cursor should be to pick up a symbol and move it.

The WIRE command can then be used to tie the desired symbol pins together. Use the blue button unless the wire is terminating at a wire intersection. Attachment of two wires to the same pin must be avoided. This will generate an error. If more than one wire must be attached then a wire extension and a wire intersection must be used.

All functions and definitions are added to the drawing as signal names. Using the SIGNAME option, the correct attachment point on the symbol is selected and the yellow button is depressed. Then the desired text is added, followed by a carriage return. The next option selection will update the screen showing the newly added signame.

There are some text symbols that will initially produce an error in a signal name. Leading parenthesis is one example. To get around this SCALD glitch the signame must be inserted without the offending symbols. Then the CHANGE option can be selected and the appropriate change commands (presented in the next section) can be used to add the missing text. There are very few occasions where this will be necessary.

Caution must be exercised when a symbol version is changed after signal names have been attached. Occasionally a signal name attached directly to a pin, rather than to a wire added to the pin, will be lost or misplaced if the version of

the symbol is changed. The SHOW ATTACH option is invaluable in checking for this problem.

Finally, two symbol pins must not be overlaid with the assumption that they are connected. All connection pins must be connected using wires or specified signal names. Also a pin must not be crossed with a wire unless it is to be connected to that pin (if it touches a pin chances are very high that it is connected to that pin). However, wires that cross each other without an intersection dot are not connected.

6. Signalname Change Commands

The following keystrokes will cause changes to any selected text. In some cases it is the only method of inserting the required text.

The CHANGE option is selected and the text is grabbed with the cursor and the yellow puck button. The text origin is at the lower left corner of the text. When the character string appears in the lower corner of the screen the following commands can be used to move the "|" symbol (| is the cursor indicator).

control f move forward one character along the string
control b move backward one character along the string
control e move to the end of the line
control a move to the beginning of the line

Deletion of the text located immediately to the right of the | cursor is accomplished through the following commands:

control d delete the character to the right of the cursor
control k delete the line to the right of the cursor
control s search for the next character or word

Text can be added to the right of the cursor by typing the additions and either continuing to edit the line or by pressing return when the change is complete. Again the change will not appear on the screen until another option is selected or another line of text is selected for change.

C. THE CONNECTIVITY FILE

The connectivity file is the file used as the MACPITTS FLOWCHART compiler's input file. This file is generated any time a logic drawing is written to a file (using the WRITE command). The drawing name is actually a directory that contains four files. The only MACPITTS usable file is the connectivity file. This file is actually called logic_cn.1.1 and the unix path name is <flowchart_logic_drawing_name>/logic_cn.1.1.. however, it will be referred to as the connectivity file throughout this document. Figure 4.4 is an example of a small connectivity file. It contains a numerical listing of all signal names, including system generated names denoting wire connections between symbol pins, and a corresponding signal number assigned by SCALD. Each of the symbols on the flowchart receives a property number when the file is saved and is listed in the connectivity file based on its property number. The corresponding pin names are then grouped in reverse alphabetical order under the property number line. Attached to each pin name is the corresponding signal number from the signal number list. MACPITTS functions assigned as signal names in the flowchart are passed to the MACPITTS compiler based on this listing.

The signal number zero is important as it designates a non-connection. In this flowchart system there should be no unconnected pins other than blank definition pins on the signalnames symbol. The flowchart compiler has an error checking routine that will point out this discrepancy if it occurs. Any uncorrected errors that were in the flowchart drawing when it was saved will be ignored in this file. However, the items influenced by those errors will also usually be ignored or in the worst case the system will work around the error modifying its list of connections accordingly. This is undesirable.

When transferring this file to the VAX or ISI machines use the ftp commands GET or PUT. Since the system net connection instructions fluctuate dramatically from quarter to quarter, specific commands will not be included here.

V. FLOWCHART COMPILER

A. CONCEPT and METHOD OF ATTACK

The main vehicle of discussion in this chapter is the flowchart shown in Figure 5.1. Most examples provided in the comprehensive explanation section of this chapter are built around this flowchart, even if it is not explicitly mentioned.

This compiler converts the connectivity file of a flowchart produced by the SCALD system into a functional MACPITTS input file. Currently the connectivity file contains three major categories of data: the signal name list. the definitions block file. and the flowchart property lists. Figures 5.2a and b provide a detailed example of this connectivity file and Figure 5.1 provides the flowchart that produced it. All of the data required for the MACPITTS code is included in the signal name list. The rest of the file provides the connection information required to arrange this signal name data in the correct order. Note that all functions. tests. actions. definitions and the title and process names are contained in the signal names list. Each is preceded by a number called the signal number. These signal numbers are also found after the pin names listed in the definition and property lists. All pins are "connected" to a signal or another pin based on this number. Each line of the connectivity file is considered a RECORD which contains one or more FIELDS. The record separator is either a carriage return or a ";" and the field separator is a quotation mark. This connectivity file is found

in the SCALD system's unix files in the flowchart's name director as the logic_cn.1.1 file. The path name is <flowchart_name>/logic_cn.1.1

Based on this information it becomes apparent that a method of searching for a number or string pattern in one or more fields of one record and matching it with other fields in another record is required. Also, once these matches are found some new combination of these fields of data must be created. This data record manipulation must continue until the final goal of a MACPITTS input file is reached. The Unix C based language AWK provides all these capabilities. AWK appears to have derived its name from the word awkward and seems to be a combination of the C. Fortran and Basic languages centered around string pattern recognition. It has definite advantages, some frustrating disadvantages and is used as the foundation language of this compiler.

The goal of this thesis was to prove the flowchart concept. This compiler will currently function only if the compiler input file conforms to the SCALD connectivity file format. This is a reasonably severe limitation, however. because of AWK's versatility any CAD tool connectivity or net file should be convertible to the SCALD format.

Figure 5.3 provides an overview of the attack method used in this compiler. There are six main blocks of code contained in the Unix directories FILENAMECHANGE. FIRST. THESISPROP. SIGSORT. FINALSORT and THESISDEF. A complete description of the compiler methodology is included in the compiler overview and specifics sections of this chapter. Problems, limitations

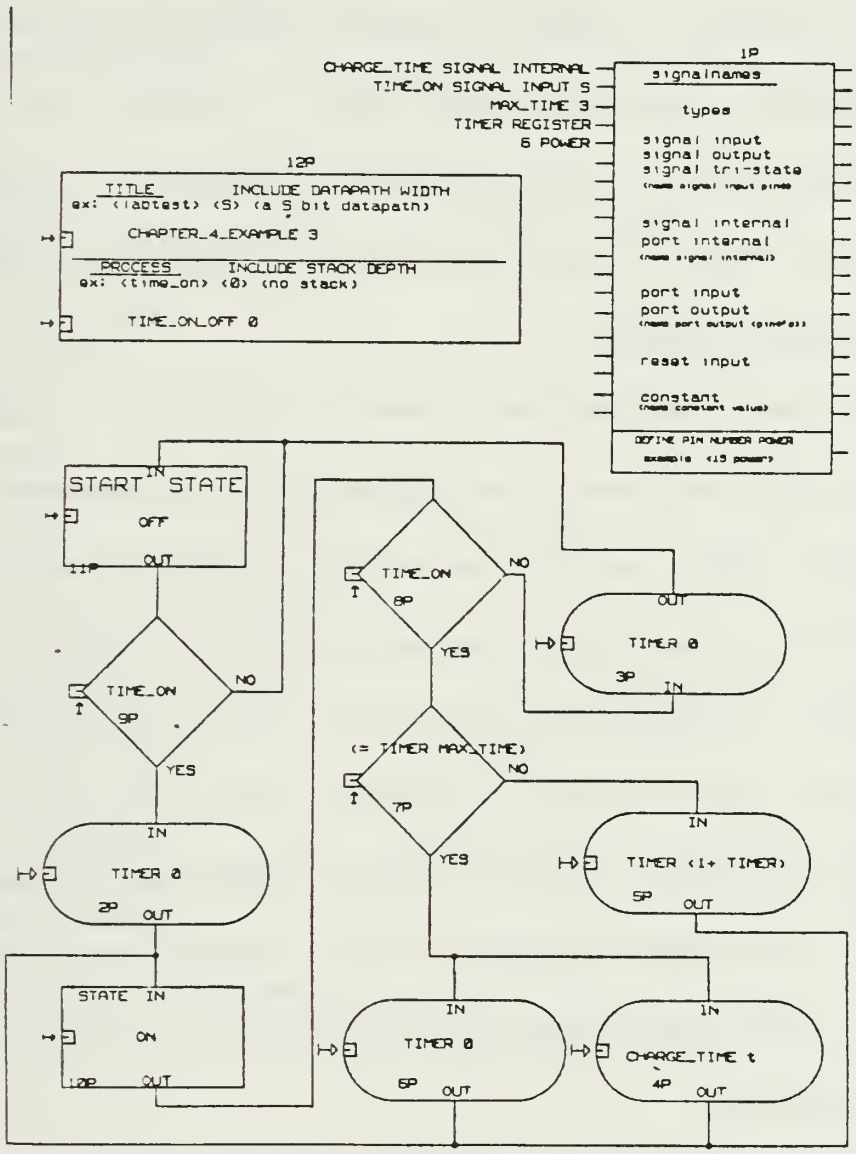


Figure 5.1 Example Flowchart Generated Using the SCALD System

and specific suggestions and recommendations for compiler improvement are also included.

B. AWK

The AWK language or compiler is usually available as one of the compiler packages included in Unix based systems [Ref. 4]. This eight page document provides a basis in, but no feel for, the language. Therefore, in an attempt to make the actual compiler code understandable AWK highlights are provided.

1. General AWK Concept

AWK starts at the first line or record of an input file and performs all searches and actions designated in the program file using a case basis approach. Once all actions within a case have been completed AWK advances to the next input data record and reattempts all searches and actions on the new line. AWK will never return to a previously evaluated record. Therefore, if a piece of data located in a later record is needed to modify an earlier record (as in coupling a signal name at the top with a pin name near the bottom) the earlier record must be stored in an array within the program or sent to a file and used in a follow-on awk program. Both of these methods are used extensively in this compiler.

a. Case Basis

The case basis analysis mentioned above is important. The user written AWK program looks for an action or pattern match in the input data file based on the action's "case" position in the program. Once an action statement

```
FILE_TYPE = CONNECTIVITY;
{GED release: 7.27TFinal:Tue Jan 14 19:48:22 PST 1986}                    "def"0;
0"NC";                                                                    "def"0;
1"TIME_ON_OFF 0";                                                         "def"0;
2"CHAPTER_4_EXAMPLE 3";                                                   "def"0;
3"OFF";                                                                   "def"0;
4"ON";                                                                    "def"0;
5"TIME_ON";                                                               "def"0;
6"TIME_ON";                                                               "def"0;
7"(= TIMER MAX_TIME)";                                                    "def"0;
8"TIMER 0";                                                               "def"0;
9"TIMER (1+ TIMER)";                                                      "def"0;
10"CHARGE_TIME t";                                                        "def"0;
11"TIMER 0";                                                              "def"0;
12"TIMER 0";                                                              "def"0;
13"6 POWER";                                                              "def"0;
14"TIMER REGISTER";                                                       "def"0;
15"MAX_TIME 3";                                                           "def"0;
16"TIME_ON SIGNAL INPUT 5";                                               "def"17;
17"CHARGE_TIME SIGNAL INTERNAL";                                          "def"16;
18"UN$1$COND$9P$a";                                                       "def"15;
19"UN$1$COND$8P$a";                                                       "def"14;
20"UN$1$COND$7P$a";                                                       "def"13;
21"UN$1$COND$8P$no";                                                      "def"0;
22"UN$1$COND$9P$no";                                                      "def"0;
23"UN$1$COND$7P$no";                                                      "def"0;
24"UN$1$COND$7P$yes";                                                     "def"0;
25"UN$1$INTSTATE$2P$out";                                                 "def"0;
26"UN$1$COND$9P$yes";                                                     "def"0;
%"SIGNALNAMES"                                                            "def"0;
"1","(3325,3750)","0","weist.wrk","1P";                                   "def"0;
;                                                                         "def"0;
;                                                                         "def"0;
"def"0;                                                                   "def"0;
"def"0;                                                                   "def"0;
"def"0;                                                                   "def"0;
                                                                          "def"0;
```

Figure 5.2a Example SCALD Connectivity File (first half)

```
%"INTSTATE"
"1","(1000,825)","0","weist.wrk","2P";
;
INTSTATE" ";
"setq"12;
"out"25;
"in"26;
%"INTSTATE"
"2","(3075,1750)","0","weist.wrk","3P";
;
INTSTATE" ";
"setq"11;
"out"22;
"in"21;
%"INTSTATE"
"1","(3225,125)","0","weist.wrk","4P";
;
INTSTATE" ";
"setq"10;
"out"25;
"in"24;
%"INTSTATE"
"1","(3175,875)","0","weist.wrk","5P";
;
INTSTATE" ";
"setq"9;
"out"25;
"in"23;
%"INTSTATE"
"1","(2200,125)","0","weist.wrk","6P";
;
INTSTATE" ";
"setq"8;
"out"25;
"in"24;
%"COND"
"1","(2200,1250)","0","weist.wrk","7P";
;
COND" ";
"zcond"7;
"yes"24;
"no"23;
"a"20;
%"COND"
"1","(2200,2075)","0","weist.wrk","8P";
;
COND" ";
"zcond"6;
"yes"20;
"no"21;
"a"19;
%"COND"
"1","(1100,1600)","0","weist.wrk","9P";
;
COND" ";
"zcond"5;
"yes"26;
"no"22;
"a"18;
%"STATE"
"1","(875,175)","0","weist.wrk","10P";
;
STATE" ";
"zgoto"4;
"outother"19;
"in"25;
%"STATE"
"3","(875,2250)","0","weist.wrk","11P";
;
STATE" ";
"zgoto"3;
"outstart"18;
"in"22;
%"TITLE"
"1","(850,3150)","0","weist.wrk","12P";
;
;
"title"2;
"process"1;
END.
```

Figure 5.2b Example SCALD Connectivity File (second half)

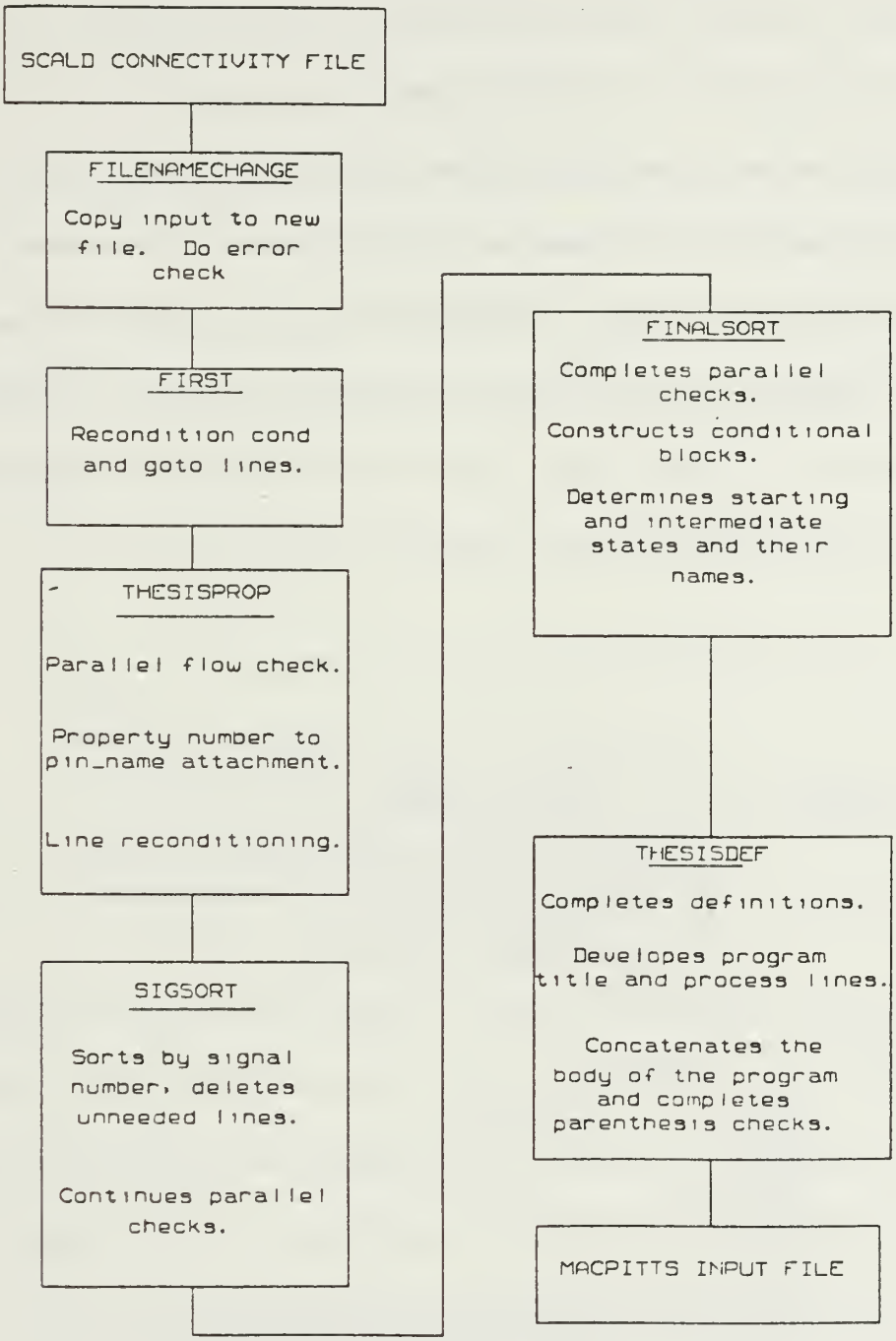


Figure 5.3 Flowchart Describing The Compiler

in the AWK routine has been completed no other search or action will be undertaken on that input data line. The following example shown in Figure 5.4 demonstrates this. The line {print $0} means transfer the current data line to the screen. If this action line were the first line in Figure 5.4, instead of the last, none of the other searches would ever be attempted even if there were matches available in the data. Again, when the first action in the AWK routine is completed AWK advances to the next data record and starts over. Therefore, the order of search in the AWK routine can be critical. Similarly, the location of a record line in the data file can also be critical.

```
   INPUT FILE                AWK ROUTINE
1"in_out":                  BEGIN { FS " ""}
2"out_in";                  $2 ~ /in/{in[i] = $1; i++}
3"outing";                  $2 ~ /out/{out[j] = $1; j++}
4"inning";                  $2 ~ /yes/{yes[k] = $1; k++}
5"ingot":                   {print $0 }
6"outward":
7"toward";                    RESULTS
                    in[1] = 1          out[1] = 6
                    in[2] = 2
                    in[3] = 3
                    in[4] = 4
                    in[5] = 5
```

Figure 5.4 AWK Routine Case Basis Example

---

b. Arrays

AWK arrays are extremely versatile. Both the name and the array index can be either numeric or string in form. Figure 5.5 provides a sample input file, the AWK command creating a concatenating string variable indexed array and the resultant array created. The field separator is a quotation mark and the record separator is a semicolon. These can be designated in the BEGIN line of the AWK routine (hereafter referred to as the "routine"). Note that field 1 is numeric while the others are not.

The AWK symbols are:

$2	Field 2
~	Search for the pattern within the delimiters / /
$2[$3]i	An array with name = field 2 indexed by field 3 of the input line
/^out$/	Look only for the string out in a field and no other characters
/out/	Look for a string containing the pattern out
$2[$3]$4	Concatenation of the current array element and the 4th field of the current record line.

One should note three major items in Figure 5.5. A string index cannot be incremented or decremented like a numeric index. The search for /^out$/ specifically delimited the second field to the word "out". If the search had been for /out/ then the array would have included the element "outside[26P] = "total"". Finally, the order of the input data file (hereafter referred to as the file) is critical in concatenating the 4th fields in a coherent manner.

```
  INPUT FILE                    AWK ROUTINE
1"out"14P"this is an":        BEGIN{FS = " " ":RS = ":"}
2"out"A"of here";             $2 ~ /^out$/{$2[$3] = $2[$3]$4
3"in"B"trouble";              {print $3,"pin name is"$2."ln number is",
4"out"c"bound";                $1 > "file1"}
5"next"A"for":
6"out"14P"array ";                 ARRAY RESULT
7"in"24P"";                   out[14P] = "this is an array example"
8"outside"26P"total":           out[A] = "of here"
9"out"14P"example";             out[c] = "bound"
```

Figure 5.5 Concatenating String Indexed Array Example

c. Action Statements

Action statements can be any desired action. Some include: if-then case constructs, array loading, output generation using print statements, variable assignments, loops etc. The while and for loop constructs are available. The loop commands will not recursively evaluate the record lines in the input file. Use them with previously loaded arrays and variables. The print action is also versatile. It allows concatenation without field separators or placement of user designated output field separators (OFS) that may be different from the input file separator. A comma indicates field separation while lack of one indicates concatenation. This is also true in any action as in the array in Figure 5.5. Output fields can be arranged in any order and variables or text can be added as

desired. The redirect symbol " > " is used to overwrite or create a file named within the quotes following it or by a variable. The " >> " symbol appends output to the file.

d. Other Useful AWK Commands and Variables

The following built in AWK commands and variables may be used as desired. Most have been used in this compiler.

NR	The number of the current record or line
NF	The number of fields in a record
FS = " "	Set the input field separator equal to that contained in " "
OFS = " "	Set " output " " " " " " " " " "
RS = " "	Set the input record " " " " " " " " " "
ORS = " "	Set " output " " " " " " " " " "
length	The number of characters in a string
split	Split a string into a number of arrays
substr	Pull a sub string out of a string field
break	Exit the AWK routine
continue	Jump to the next evaluation in a loop
next	Jump to the next input record
FILENAME	The current input file name

It is important to note that if the user designated record separator (RS) is not a carriage return and the file contains carriage returns at the end of each line then this carriage return will become the first field of the next record. This can cause confusion in identifying the correct field number of a string in a record line since the first field appears to be invisible. The FS, RS, OFS, and ORS can be set anywhere, but normally they are set in the BEGIN statement of the routine. The BEGIN and END statements allow actions to be conducted before and after the file is evaluated. All actions in the routine are designated by the { } brackets enclosing the action. An action is considered completed when the closing bracket

matching the opening bracket is encountered. This allows multiple actions to occur within a single action statement.

C. DETAILED COMPILER OVERVIEW

This compiler is again based on the SCALD system connectivity file format (Figure 5.2). In reviewing the connectivity file each symbol on the flowchart is identified by a property number and related pin names as discussed in chapter IV. Each pin name line contains a signal number which refers to a signal name at the top of the file. There are two types of pins; a function, test, or action pin and an intersymbol connection pin. The conditional test pin is labeled zcond while the action pins are labeled zgoto and setq. The intersymbol connection pins are labeled yes, no, in, out, outstart, and outother.

The steps required to get from the connectivity file to a MACPITTS input file are:

(1) Copy the input file for compiler use and check for obvious errors.
(2) Precondition the zcond and zgoto function pin names.
(3) Attach symbol property numbers to symbol pin names for identification.
(4) Attach functions to function pins (setq, goto, cond) by signal number.
(5) Attach state labels to their symbol output pins.
(6) Attach conditional test functions to their symbol yes pins
(7) Rearrange and sort the new file in ascending signal number order.
(8) Determine parallel action symbols
(9) Build action-transition blocks .
(10) Combine these action blocks with conditional yes and no pins
(11) Develop case basis or series conditional constructs
(12) Determine start state label and attach all appropriate conditionals and actions.
(13) Determine all other state labels and attach their conditionals and actions.
(14) Concatenate all state constructs in the final MACPITTS input program code.
(15) Construct the title and process lines required for MACPITTS
(16) Construct the MACPITTS definitions block (clock, power, signals, etc.)

(17) Concatenate the title, definitions, process, MACPITTS program code.
(18) Complete the parenthesis requirements for MACPITTS

The next section of this chapter and Appendix B cover the compiler in more detail. For maximum understanding it is recommended that a file be run through the compiler after all the command files have had the data file deletion lines commented out. This will allow a printout of all command files, AWK routines and the generated data files to be reviewed. Each directory command file provides the order and names of all data files created by the compiler.

D. COMPILER SPECIFICS

The compiler code included in Appendix B is well documented, however, the next few paragraphs will attempt to show the reasons behind the development of each stage of the compiler. Again Figure 5.3 gives an overview of the goals of each directory of AWK routines. The paragraphs below are divided into these directories in their executional order. The main directory is a subdirectory called DOIT. All other directories are subdirectories of DOIT. This allows a user to stay in the top level directory and enter the command "Compileflowchart". Results will appear in this top level directory with a dot mac extension. The user is always at least one level away from the compiler routines which should prevent unwanted file accumulation and modification of the compiler.

1. Doit

This is the main command file. It calls all sub directories and runs the sub directory command files. It also shows the sequence of directory calls and

contains user interactive routines. This allows the user to either kill the compiler or continue if initial errors are detected in the input file. It is written in C SHELL command form for the unix system [Ref. 4].

2. Filenamechange

The goal of this directory is to create a copy of the user's input file for the compiler's use and check it for obvious errors. The request for an input file name is required because the passage of variables between command files is currently beyond the author's capability. Since the compiler will attempt to compile a named file, whether it exists or not, the user is interrogated after his file name is entered to verify the correctness of the entry.

The Noconnectck.cmd file checks for any pin that is not connected to another pin or signal based on the connectivity file's signal number 0. This is the no connection indicator. Every pin must be attached to a signal number other than 0 except for the definitions (def) pins and the process name pin. It uses the same methods to detect missing blocks or symbols such as the title, start state and definitions blocks. If an error is found it is sent to the error.dat file. The Errorout.cmd routine then sends the error messages to the screen. Control is returned to DOIT after all generated data files have been purged from the directory to save space and prevent possible errors during the next compiler run.

3. First

The Doit command file then runs the directory FIRST which conditions the newly copied connectivity file. To guarantee that the two pin names "goto"

and "cond" were at the top of their pin name list in the connectivity file the letter "z" was added to each producing "zcond" and "zgoto". This was based on the inverse alphabetical ordering of the file by SCALD as discussed in chapter IV. These z's must now be removed without altering the rest of the file. This requirement was generated when the author was mid way through the development of the next directory and was added to the system at that time. Awk's inability to move up in a file was the driving factor (the cond and goto lines had to appear prior to any other pin names in their property number group). The output file is thesis.dat. Control is returned to Doit after all unneeded files are purged.

4. Thesisprop

The command file then calls the sub command file Assign in the THESISPROP directory which accomplishes six file conversions using six AWK routines. Propassign.cmd attaches the individual symbol property numbers to their respective pin names to allow deletion of excess data lines in the file without loss of symbol identification. Setqcond.cmd attaches the functions listed in the signal name block to the setq, goto and condition (cond) pin name lines base on signal number matches. Then Inassign.cmd attaches the setq and goto action and transition lines to their respective symbol "in" connection pin name lines based on property number matches. Outassign.cmd attaches the MACPITTS state labels or names to their symbol output pins (outstart and outother). Condassign.cmd attaches the conditional test function pin lines to their respective "yes"

connection pins because any conditional test is followed by the true action statement in MACPITTS. Finally, alinepull.cmd pulls the "in" and "a" pin name lines to the top of the file to assure that these lines will now be at the top of their block when the file is sorted by signal number.

As noted in chapter III and IV the property number (the symbol identity) is designated by a number and the letter P. The signal number is designated by a number. An example of this directory's output file (property4.dat) based on the flowchart in Figure 5.1 is shown in Figure 5.6. The pin names have gained their appropriate functions, actions, and transitions and have retained their property and signal numbers. These will be required in the FINALSORT routines covered later in this chapter. This determination of what fields of information had to be retained wasn't made until well into the development of the FINALSORT directory. At that time the author backtracked and included them in the output files of this directory.

Again all unneeded files are deleted and control is passed back to the Doit command file.

5. Sigsort

The file will be sorted by signal number since all functions are now attached to their individual symbols (property number blocks) and the remainder of the compiler is based almost entirely on the connection pin interconnects (signal numbers). Doit passes control to the SIGSORT directory which sorts a maximum of 100 signal numbers. This number can be easily increased as noted in

```
> > > APULL1.DAT FILE < < < < < < < < <
2P"in"(setq TIMER 0)"26";
3P"in"(setq TIMER 0)"21";
4P"in"(setq CHARGE_TIME t)"24";
5P"in"(setq TIMER (1+ TIMER))"23";
6P"in"(setq TIMER 0)"24";
7P"a"20"";
8P"a"19"";
9P"a"18"";
10P"in"(go ON))"25";
11P"in"(go OFF))"22";
>>>>>>>>>> APULL2.DAT FILE <<<<<<<<<<;
>>>>>>>> PROPERTY4.DAT FILE <<<<<<;
""""";
""""";
""""";
{GED release: 7.27TFinal:Tue Jan 14 19:48:22 PST 1986}"0""";
1"TIME_ON_OFF 0""";
2"CHAPTER_4_EXPAMPLE 3""";
3"OFF""";
4"ON""";
5"TIME_ON""";
6"TIME_ON""";
7"(= TIMER MAX_TIME)""";
8"TIMER 0""";
9"TIMER (1+ TIMER)""";
10"CHARGE_TIME t""";
11"TIMER 0""";
12"TIMER 0""";
13"6 POWER""";
14"TIMER REGISTER""";
15"MAX_TIME 3""";
16"TIME_ON SIGNAL INPUT 5""";
17"CHARGE_TIME SIGNAL INTERNAL""";
18"UN$1$COND$6P$a""";
19"UN$1$COND$8P$a""";
20"UN$1$COND$7P$a""";
21"UN$1$COND$8P$no""";
22"UN$1$COND$9P$no""";
23"UN$1$COND$7P$no""";
24"UN$1$COND$7P$yes""";
25"UN$1$INTSTATE$2P$out""";
26"UN$1$COND$9P$yes""";
2P"(setq TIMER 0)""";
2P"out"25"";
3P"(setq TIMER 0)""";
3P"out"22"";
4P"(setq CHARGE_TIME t)""";
4P"out"25"";
5P"(setq TIMER (1+ TIMER))""";
5P"out"25"";
6P"(setq TIMER 0)""";
6P"out"25"";
7P"(cond ((= TIMER MAX_TIME)"";
7P"yes"(cond ((= TIMER MAX_TIME)"24";
7P"no"23"";
8P"(cond (TIME_ON"";
8P"yes"(cond (TIME_ON"20";
8P"no"21"";
9P"(cond (TIME_ON"";
9P"yes"(cond (TIME_ON"26";
9P"no"22"";
10P"outother"ON"19";
11P"outstart"OFF"18";
12P"title"2"";
12P"process"1"";
END.""""";
""""";
""""";
""""";
```

Figure 5.6 Example of the Property4.dat Output File

the code documentation in Appendix B. To keep the compiler based on a common language AWK was used for the sort. This is probably slow and cumbersome and a language such as GREP should be investigated for better sorting efficiency. The first routine (numberofsig.cmd) determines the maximum number of signals and modifies the second command file (Sigsort1) to reduce the sort time requirement. The results are in the Sortedsig file. An example based on Figure 5.1 is shown in Figure 5.7. All unneeded files are purged and control is passed back to Doit.

```
> > > > > SORTEDSIG a data FILE < < < < < < < <;
12P"process"1";
12P"title"2";
9P"a"18";
11P"outstart"OFF"18";
8P"a"19";
10P"outother"ON"19";
7P"a"20";
8P"yes"(cond (TIME_ON"20";
3P"in"(setq TIMER 0)"21";
8P"no"21";
11P"in"(go OFF))"22";
3P"out"22";
9P"no"22";
5P"in"(setq TIMER (1   TIMER))"23".
7P"no"23";
4P"in"(setq CHARGE_TIME t)"24";
6P"in"(setq TIMER 0)"24";
7P"yes"(cond ((= TIMER MAX_TIME)"24";
10P"in"(go ON))"25";
2P"out"25";
4P"out"25";
5P"out"25";
6P"out"25";
2P"in"(setq TIMER 0)"26";
9P"yes"(cond (TIME_ON"26";
```

Figure 5.7 Example of the Sortedsig Output File

6. Finalsort

Final sorting and construction of the program section of the MACPITTS input file is accomplished by this directory. It is the most complex directory and most changes made to the compiler in the future will occur here. Six major operations must be completed. These are: make final parallel action checks and construct the action statements: construct the action transition statements; combine the conditional test and action transition statements; develop the case basis and serial conditional constructs; determine state labels and build each of the states from the conditionals and action blocks already constructed; and finally arrange all states in the correct order for MACPITTS.

Parain1.cmd evaluates the sortedsig file (see Figure 5.7) for parallel action statements based on matching signal numbers on "in" connection pins. Since parallel actions in a conditional MACPITTS statement are basically a serial string listing of the actions, these parallel actions are concatenated into a single string using a concatenating array approach. Parallel action without a conditional (setq's only) can be forced using the "par" function in MACPITTS. This par function is included in all state blocks since serial setq actions are currently not allowed in this flowchart system. This section of the compiler also cannot yet tell if the parallel action statements constructed here as a serial string will be included in a conditional, hence the use of par.

Outin.cmd and Inout.cmd attach the transition statements to the end of each of the action blocks. This was done after all parallel actions were evaluated

to prevent multiple transition statements in a single action statement. It is also assumed that every action must transition to a new state (or restart in the same state). Therefore, each action must include a transition. Note also that transition to two different states from one action statement is invalid because only one state can be active at a time within a process in MACPITTS.

These action transition lines are then attached to the yes and no conditional connection pins based on signal number matches between action "in" pins and the "yes/no" pins. Yesin1.cmd accomplishes this task. Note that the conditional test had already been added to the yes pin earlier in the THESISPROP directory. Therefore, the conditional test and true action transition statement is complete. Parenthesis are added to comply with MACPITTS form. In MACPITTS the "no" portion of a conditional is always true, but is isn't acted upon unless the yes portion fails. This is another example of the case basis construct. To indicate this in MACPITTS a "(t " is added as the test header of the action statement of a conditional no pin. The complete conditional is now available in blocks of yes and no connected lines.

The dual conditional construct is now evaluated by the ayes.cmd routine. There are two types of these dual or serial conditionals: the standard case basis where the second conditional is connected to the no pin of the first and the if-then-if construct when the second conditional is connected to the yes pin of the first. These are easier to see in Figure 5.8 and fit the standard algorithmic language as follows:

CASE BASIS

```
if then (action)
else if then (action)
else (action)
```

SERIAL BASIS

```
if then (if then (action)
        else (action)
else (action)
```

In both cases the maximum depth of conditionals is currently limited to two. This can be extended and approaches for doing this are presented later in this chapter and in Appendix B. Existence of these two constructs is evaluated and a block of conditional statements is created and denoted by a keyword "doublecond" for later identification when the states are built.

The header.cmd routine constructs the final program portion of the MACPITTS input file using all of the pieces constructed earlier in this directory. All of these conditional and action transition statements are loaded into three large arrays. Then the file is searched for the state labels attached to the two state output pins (outstart and outother). The outstart label and all the conditionals and actions attached to that pin by signal number matches are sent to one file. The outother labels and their corresponding conditionals and actions are sent to another. This guarantees that the start state appears first in the program portion of MACPITTS when the "other" states are appended to the bottom of the start state file.

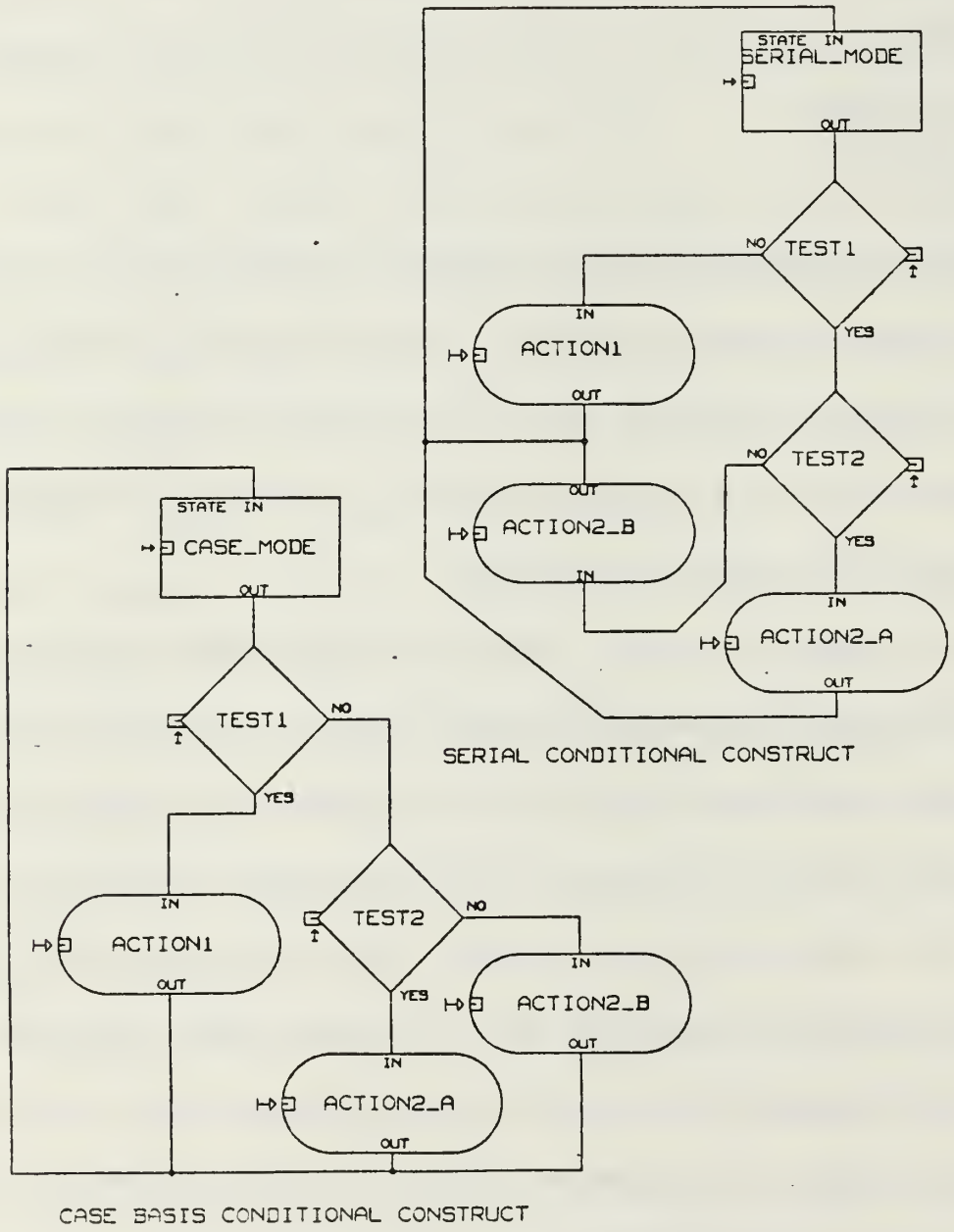


Figure 5.8 Example of the Two Types of Conditional Constructs

```
finalin"3P"(setq TIMER 0)(go OFF))"21
finalin"11P"(go OFF))"22
finalin"5P"(setq TIMER (1+ TIMER))(go ON))"23
finalin"6P"(setq CHARGE_TIME t)(setq TIMER 0)(go ON))"24
finalin"10P"(go ON))"25
finalin"2P"(setq TIMER 0)(go ON))"26
(cond (TIME_ON"doublecond"8P";
   (cond ((= TIMER MAX_TIME)(setq CHARGE_TIME t)(setq TIMER 0)(go ON))"doublecond";
      (t (setq TIMER (1+ TIMER))(go ON)))"doublecond";
   (t (setq TIMER 0)(go OFF))))"doublecond";
signalend""8P";
checked"8P"yes"(cond (TIME_ON"20";
checked"8P"no"    (t (setq TIMER 0)(go OFF)))";
checked"9P"no"    (t (go OFF)))";
checked"7P"no"    (t (setq TIMER (1+ TIMER))(go ON)))";
checked"7P"yes"(cond ((= TIMER MAX_TIME)(setq CHARGE_TIME t)(setq TIMER 0)(go ON))";
checked"9P"yes"(cond (TIME_ON(setq TIMER 0)(go ON))";
>>>>> SORTEDSIG a data FILE <<<<<<<<;
12P"process"1";
12P"title"2";
9P"a"18";
11P"outstart"OFF"18";
8P"a"19";
10P"outother"ON"19";
7P"a"20";
8P"yes"(cond (TIME_ON"20";
3P"in"(setq TIMER 0)"21";
8P"no"21";
11P"in"(go OFF))"22";
3P"out"22";
9P"no"22";
5P"in"(setq TIMER (1+ TIMER))"23";
7P"no"23";
4P"in"(setq CHARGE_TIME t)"24";
6P"in"(setq TIMER 0)"24";
7P"yes"(cond ((= TIMER MAX_TIME)"24";
10P"in"(go ON))"25";
2P"out"25";
4P"out"25";
5P"out"25";
6P"out"25";
2P"in"(setq TIMER 0)"26";
```

Figure 5.9 Example of the Headerck.dat file (Midpoint File in FINALSORT)

Figure 5.9 is an example of the input file to the header.cmd routine based on Figure 5.1. There are three keyword blocks: "doublecond" for double condition statements. "checked" for single condition statements. and "finalin" for action only statements. These are loaded into the arrays mentioned above. Note that the actions and single conditional statements are duplicated in the doublecond lines. It is imperative that the states be built in the correct sequence. First a double conditional match is attempted, then a single conditional and finally an action only match is attempted. This guarantees there will be no duplication since AWK moves to the next data line when a match and its action are completed. Figure 5.10 provides an example of the final output of this directory. It is the program portion of the MACPITTS input file based on the flowchart in Figure 5.1. Any modifications to the compiler must take this ordering into account. Again suggestions for changes in this area are covered in Appendix B.

7. Thesisdef

The last requirement is to construct the program title. the definitions block and the process identification line and then append the program block constructed by FINALSORT and complete parenthesis checks. This is relatively simple and is accomplished by the THESISDEF directory using the compiler's copy of the original connectivity file. All the information is available in the signal name block and the def and title property blocks. The final MACPITTS input file is constructed by concatenating the title. definitions, process and program

```
OFF
 (par
   (cond (TIME_ON(setq TIMER 0)(go ON))
         (t (go OFF)))

 )
ON
 (par
  (cond (TIME_ON
      (cond ((= TIMER MAX_TIME)(setq CHARGE_TIME t)(setq TIMER 0)(go ON))
          (t (setq TIMER (1+ TIMER))(go ON)))
       (t (setq TIMER 0)(go OFF))))

 )
```

Figure 5.10 Example of a Finished MACPITTS Program Portion File

block and adding two parenthesis at the end to account to match the open parenthesis in the title and process lines. All other parenthesis were handled during the program block buildup in the other directories. An example of a completed MACPITTS file based on Figure 5.1 is shown in Figure 5.11.

E. LIMITATIONS

The compiler has several limitations in its current form which include:

(1) A restricted and reasonably odd symbol function syntax.
(2) A maximum of 100 signals.
(3) A maximum of two conditionals in a case construct.
(4) Non-implementation of the MACPITTS subroutine capability.
(5) A limit of one process per flowchart.

```
(program CHAPTER_4_EXAMPLE_3

(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def 6 power)
(def TIMER register)
(def MAX_TIME 3)
(def TIME_ON signal input 5)
(def CHARGE_TIME signal internal)

(process TIME_ON_OFF 0

OFF
 (par
   (cond (TIME_ON(setq TIMER 0)(go ON))
         (t (go OFF)))

 )
ON
 (par
  (cond (TIME_ON
      (cond ((= TIMER MAX_TIME)(setq CHARGE_TIME t)(setq TIMER 0)(go ON))
          (t (setq TIMER (1+ TIMER))(go ON)))
       (t (setq TIMER 0)(go OFF))))

 )
))
```

Figure 5.11 Example of a Finished MACPITTS Input File

(6) Non-implementation of serial action within a single state.
(7) Restriction to a SCALD produced input file format.

Each of these areas will be discussed and suggested solutions will be provided.

1. Function Syntax

The syntax required for function entry on the flowchart in a conditional or action symbol is discussed in detail in chapter III. It is basically in a LISP format and requires the user to learn an odd syntax. This is required because the compiler basically rearranges the functions as an unchangeable entity. The form and content of an entered function is not changed. An example should clarify this discussion. The user wishes to test for equality between two integer variables. One variable is a function of two other variables. A reasonable syntax might look like this:

a = (b + c) or possibly a == (b + c)

However, the syntax required by the compiler for this function is:

(= a(b + c))

All the parenthesis are required. The SCALD system must be fooled into allowing these parentheses by use of the ged change command (chapter IV). This reasonably simple function has become difficult to enter and understand solely based on its syntax.

One or more AWK routines could be written as a pre-processor to this compiler to process the signal names block of the connectivity file. This pre-processor would search for "normal syntax" statements and rearrange them into

the correct LISP type syntax required by the compiler. More complex functions would require more processing. Making this change would require two major steps. First a new and simpler syntax format would have to be designed that covered all of the MACPITTS capabilities. This would be a major task considering the complexity of MACPITTS. Growth considerations would also have to be considered since the MACPITTS compiler has been dissected well enough by LtCdr MALAGON-FAJAR to allow major additions to MACPITTS. The precompiler processing code would then have to be written and rigorously tested to assure reasonable reliability in the conversion.

2. Maximum Number of Signals Allowed

The current limit on the number of individual signals is 100. This is based on the idea that 100 functions, actions, and interconnections would make a rather large flowchart. Since the main thrust of this thesis was to provide a feasibility demonstration of the concept, this limit was reasonable. However, it can be easily increased by adding more sort routines to the SIGSORT directory and revising the Sigsort1 command file to accommodate them. Also the loop limit located in the END statement of the parain1.cmd file located in the FINALSORT directory would have to be increased.

3. Conditional Case Evaluation Limitations

MACPITTS has no limit on the number of case conditionals that may exist in a single state. However, as discussed in chapter II these case conditionals are all evaluated in parallel. The chip size will increase dramatically as the

number of parallel evaluations increases due to the current layout algorithms. If the actual function evaluated in each conditional is complex then the chip size will grow proportionally.

The current limit of two conditionals in series is based on the limitations imposed by the FINALSORT directory routines. Conditional construct buildups are limited by the ayes.cmd routine and the header.cmd routine. The ayes.cmd routine has a detailed recommendation for improvement included in its documentation in Appendix B. Essentially several new buildup routines would be required that could key on the "no" pin names listed in the acheck.dat file. Case constructs are indicated by a lack of an action transition statement attached to these "no" pin names. A connection signal number is found in its place. It is recommended that a multiple case conditional flowchart be constructed and run through the compiler. Careful study of the resultant acheck.dat and headerck.dat files should provide a modification direction for the user. The rm lines in the Finalsort command file will have to be commented out to allow retention and printout of the data (.dat) files.

4. MACPITTS Subroutine Implementation

The MACPITTS subroutine capability has not been implemented in this flowchart system. This is not to say the compiler cannot handle this subroutine capability. This is because MACPITTS documentation is not available on the syntax and procedures required to implement subroutines. It is assumed that the syntax "call<subroutine_name>" would replace the current "go<state_label>"

statement. If this assumption is correct then a new symbol could be established for this purpose. Modification of the FINALSORT directory would be required and an evaluation of changes in the preceding directories would have to be conducted to ensure the new symbol wasn't eliminated as an unneeded line in the input file. It is recommended that the correct MACPITTS format and syntax be determined as a precursor to modification of the compiler.

5. Single Process Computation

Currently only one MACPITTS process block can be constructed at a time by the compiler. MACPITTS, of course, is capable of handling multiple. parallel running, processes in its input file. The solution to this limitation is straightforward.

Within the Doit command file located in the DOIT directory the user could be interrogated interactively for multiple flowchart file names. This assumes that each process in created by a different flowchart. A final input flowchart would contain the title block and the signal definitions blocks for all the processes. Multiple runs of the compiler would then allow buildup of the individual process blocks separately. The title block symbol would have to be modified (another version) to allow just the process information to be attached to the individual flowcharts. Major revision of the THESISDEF directory routines would be required to get all the processes concatenated properly, however. this directory is the least complex of all the directories in the compiler.

6. Serial Action Implementation

MACPITTS is capable of handling serial actions (no conditionals) in a single state. One action is accomplished during each clock cycle. This compiler cannot construct the MACPITTS code in this manner. However, this is not considered a limitation because an action transition statement is completed in a single clock cycle just as the serial action statement. The action transition statement is evaluated in parallel so there is not clock speed degradation and the chip area differences appear to be minimal. A detailed analysis of this phenomenon has not been completed, however, so one may assume that there are some differences in the two approaches. Currently, modification of the compiler is not considered a requirement.

7. SCALD System Format Requirement

As noted previously, the compiler will correctly compile a file only if it conforms to the SCALD system's connectivity file format. This effectively eliminates other CAD systems, however, the systems currently available at NPS do not have the detailed capabilities of SCALD.

Considering the capabilities of AWK, converting another system's net output files into a format essentially the same as the SCALD's should not be difficult. Competing system net files were not available for detailed comparison, therefore, this assumption may be invalid. In the author's opinion the assumption is valid.

F. COMPILER SUMMARY

This compiler has demonstrated the feasibility of the flowchart-to-chip concept presented. It has several limitations as previously discussed, however, with only minor changes these limitations could be overcome. If an attempt were mounted to "start over" it is recommended that the LISP language be investigated as the source language of the compiler instead of AWK. This opinion is based on the flexibility of LISP and not on the drawbacks of AWK. This shift in source would entail considerable def struct development (these def struct's are LISP structures used to manipulate data structures). Field and record detectors, output manipulators, character string recognition structures and array structures would all be required. AWK provided these capabilities in a general manner. Specific structures in LISP built for specific input file manipulations may be much more efficient, however, more development time will be required.

The next major research direction of this flowchart system when tied to MACPITTS should be the evaluation of compiling the flowchart into the MACPITTS object code file instead of the current input file format. This may provide a less restrictive syntax capability. It has not been investigated in this thesis. If object code can be generated then the possibility of separately manipulating the data path and control structure of MACPITTS may develop. If this occurs then the flowcharting system should be adjusted to allow separate development of the data flow and control structures. New symbols and methods of inserting functions and actions would most likely be required.

VI. CONCLUSIONS

A. SUMMARY

This thesis has presented a flowchart-to-chip concept. To do this, a flowcharting system and a compiler capable of converting the flowchart into a MACPITTS input file has been developed. MACPITTS is then called upon to generate the requisite VLSI chip layout for completion of the concept.

Limitations of this flowchart compiler and the MACPITTS compiler have been found and solutions to the flowchart limitations have been presented. Generation of solutions to the MACPITTS problems is left to other researchers.

The final results of this thesis are encouraging. Actual complex flowcharts were constructed and successfully compiled into correct MACPITTS input file code. An unexpected benefit introduced by the current limitations of the flowchart compiler have forced a compromise between the chip area required to implement the layout and the clock speed. Both appear to have been optimized in most respects. The flowchart system is relatively easy to use and requires a relatively short learning curve when compared to the MACPITTS system.

The results of this Flowchart-To-Chip concept might appear to be underwhelming when considered in the rather limited perspective provided by silicon compiler technology. However, when one considers that this flowchart compiler actually converts a flowchart into a LISP language program the.

possibilities of extended use in general computer programming become more evident and the results gain new meaning.

B. RECOMMENDATIONS

Numerous recommendations for improvement in the actual flowchart compiler used in this thesis have been presented in chapters four and five and in Appendix B. As MACPITTS and other silicon compilers are expanded this flowchart can and should be expanded to meet the capabilities provided.

MACPITTS operations that are still not fully understood are in the areas of serial-parallel flow (implemented using unnamed sub-states within a named state), and in the development and use of subroutines. Both of these areas need more investigation before implementation in the flowchart system is possible.

Further research should be considered in the general application of this flowchart capability. As languages become more complex this type of general language may become even more applicable. If the compiler were to be expanded into a multi-language translator then the positive attributes of each language would become available to the user without the limitations of learning the language. This is obvious in all current compilers which change their heirarchical input into machine code. Applications in the new AI field may make their strange languages and constructs more understandable and, therefore, more accessible.

VII. APPENDIX A

SCALD NOTE INDEX

1.0 Clock Driver for Simulator
2.0 Restore a Drawing
3.0 Unix Notes (printing, transferring files, macros)
3.1 Unix Notes (setup, remote login, disk space)
4.0 Forbidden GED Names
5.0 Change Command
6.0 Simulate Troubleshooting
7.0 Timing Verifier
7.1 Timing Verifier
7.2 Timing Verifier, page 1 of 2
7.3 Timing Verifier, page 2 of 2
8.0 Scald Troubleshooting (continuous script file, EOF error, terminal lockup)
8.1 Scald Troubleshooting (no hardcopy, swap space error)
9.0 GED Notes (show net, open collector)
9.1 Merges
9.2 GED Notes (taps, body glitch, body note)
9.3 GED Notes (signame, size, note, constant values
9.4 GED Notes (use of puck buttons)
10.0 Creating a Body, page 1 of 2
10.1 Creating a Body, page 2 of 2
11.0 Plottime
12.0 Simulator Update
12.1 Simulator Notes (simulate data entry and script files)
13.0 Loading Memory Devices

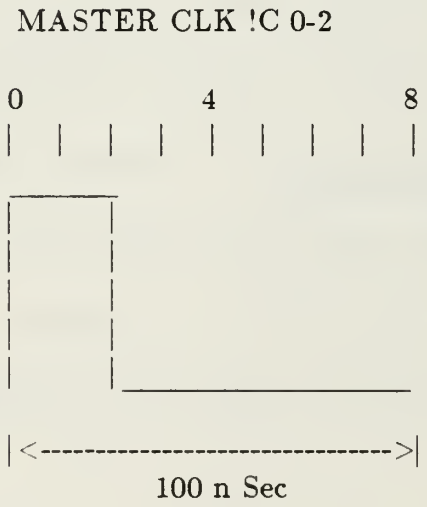
SCALD NOTE 1.0

Providing a Clock Driver for the Simulator

The CLOCK_PERIOD integer sets the period of the clock used by the simulator. Any signal with a "C" or "P" name property (e.g., MASTER CLK !C 0-3) has its behavior specified in terms of this period which is in units of nanoseconds. If unspecified, the simulator sets the period to 100 nS. The clock period must be an integer and may be changed during simulation using the PERIOD command.

The CLOCK_INTERVALS integer sets the number of evenly spaced sub-periods within the clock period. For example, if there are 8 sub-periods and the period of the clock is 100 nS, then MASTER CLK !C 0-2 is high from 0 nS to time 25 nS and low from 25 nS to 100 nS. If unspecified, the clock is divided into 10 sub-periods. The clock interval must be an integer and may be changed during simulation using the INTERVAL command.

```
simulate.cmd
    CLOCK_PERIOD  100 ;
    CLOCK_INTERVALS  8 ;
```



SCALD NOTE 2.0

To Restore a Drawing

For example, to restore the drawing Lab1:

In ged type the following lines, each followed by a carriage return

IGNORE GROUP.WRK

YES

USE RESTORE.WRK

EDIT RESTORED1
this assumes the drawing (directory) restored1 relates to Lab1. You can have many restoredN drawings (directories) retained relating to drawings other than Lab1. Look at the drawing being written to the screen. Is it the Lab1 drawing you desire? If not see what other restoredN drawings exist by issuing the DIR command in ged and then issuing the EDIT *restoredN* command again.

DIA LAB1

IGNORE RESTORE.WRK

YES

USE GROUP.WRK

SCALD NOTE 3.0

Unix Notes

Printing of files:

Scald1: cat filename > /dev/vprint
Scald2: cat filename > /net/scald1/dev/vprint

Transferring files between Scalds:

From 1 to 2:

cp filename /net/scald2/u0/loginname/group_#/filename

From 2 to 1:

cp filename /net/scald1/u0/loginname/group_#/filename

note: cp sourcepath destinationpath

Transferring all files in a directory (e.g. a drawing directory):

1) Make a drawing directory in the destination directory using:
   mkdir dirname
2) You must be in the source directory that has the drawing directory name,
   not in the destination directory when you issue the commands in step 3.
3) Between scalds:
   cp dirname/* /net/scald#/u0/loginname/group_#/dirname
   Same scald:
   cp dirname/* /u0/loginname/group_#/dirname
   note: cp sourcepath destinationpath

To make an executable macro file:

1) Use vi to create your file and save it.
2) Use chmod +x filename
3) Now while in unix any time you type your filename and carriage return,
   the unix system will execute the command lines in that file in order.

SCALD NOTE 3.1

Unix Notes

using setup:

The unix command **setup** is used to set the unix display from the default. It is most useful in being able to not have the entire unix screen update with every line that is displayed. This causes slow displaying of text lines when the unix screen is full.

In unix, issue the following commands:

setup	(this displays a menu)
backspace key	(this gets the cursor to the scrolling increment menu option)
-	(this sets the scrolling increment to the max window size)
spacebar	(this exits the setup menu)

remote login:

Usage of this unix command is **rlogin** *hostname*
For instance, if you are in scald and want to login to scald1 temporarily,
issue the unix command **rlogin** *scald1*
then issue the unix command **pwd** to see what your present working directory is in scald1. Notice that the response is slow. To quit the remote login, issue the unix command **logout** and you will be returned to scald2.

disk space:

To see what disk space is available issue the unix command **df**
Of particular interest is what space is available on /dev/rim0b

SCALD NOTE 4.0

Forbidden GED Filenames

and	or	xor
buf	ts buf	mux
2mux	4mux	8mux
reg	reg rs	reg rs comp
latch	latch rs	latch rs comp
jk	res	pass transistor
uni pass transistor	counter	shift register
identity	1 of 8 decoder	8 bit decoder
priority encoder	8 bit prio encoder	alu
adder	carry save adder	comparator
lookahead	memory	parity
inv	2and	2or
exor	dff	capacitor
inductor	lsource	lsource(I)
lsource(V)	njfet	nmos
npn	pjfet	pmos
pnp	resistor	transmission
vsource	vsource(I)	vsource(V)

Any TTL, ECL, LSTTL, F, or MEMORY chip name.

19 Feb 1986 L. Weist D. Schaeffer

SCALD NOTE 5.0

Change Command

To change a signal name, etc.

1. Select CHANGE on the ged menu or type it on the keyboard in ged.

2. Using the cursor mouse select the item you want changed.

3. When the character string shows up at the bottom left of the screen use the following control characters to move the | indicating the cursor position.
   - control f move forward one character
   - control b move back one character
   - control e move to end of line
   - control a move to the beginning of line
   - control d delete the character to the right of |
   - control k delete line to the right of |
   - control s search
   - control v go to vi to edit shift zz to return

4. Once you have the line modified as you want then you can add text to the right of the | by typing your additions.

5. Press return on the keyboard and the change is complete.

6. The change will not appear on the screen drawing until you select another function (i.e.. another CHANGE, WINDOW, etc.)

SCALD NOTE 6.0

Simulate Troubleshooting

1. Signals don't get through during simulation run.
   a. Go back to ged editor and edit your file as neccessary.
   b. Use the following ged commands in order:
      CHECK (RF5)
      ERROR (RF6)
      SHOW ATTACH (LF4)
      if you get no errors and everything looks good then:
      1. Select the ged command MOVE
      2. Using the mouse, select each merge in turn for the following:
         a. You should see the merge move and the wires connected to it should remain hooked up both to the merge symbol and the chips and wires (i.e. both ends of a wire stay attached at the end points).
         b. note, if a chip body moves with the merge, the merge is connected wrong and must be deleted and reattached. It is actually connected to the body drawing of the chip and not the I/O wire you wanted. You cannot hook a body to any other body without a wire between them. An example is you cannot overlap a merge pin and a chip pin. They do not connect. Spread them out and wire them together.
      3. Check the individual chips next and watch for moving wires that should not move.
      4. If you used AUTODOT, check for a connection that should not be dotted.

2. If all the above fails then:
   a. Re-compile the drawing for logic in unix.
      Note, you must re-compile for logic. The compiler for each mode (logic,sim,time....) overwrites the 3 compiler files.
   b. Print the file cmpexp.dat out and look for problem areas.

21 Feb 1986 L. Weist D. Schaeffer

SCALD NOTE 7.0

Timing Verifier

The timing verifier has nothing to do with simulate! To use the timing verifier insure that the verifier.cmd file is correct and also insure that the td.cmd file is what you want for the appearance of the timing diagram.

1. Issue the unix command **compile** *drawingname* **time** . errors will be documented in the file cmplst.dat

2. Issue the unix command **verify** . errors will be documented in the file tvlst.dat

3. Issue the unix command **plottime**

4. Move to the ged window and issue the following ged commands:
   IGNORE GROUP.WRK
   YES
   USE GROUP.WRK

6. Issue the ged command EDIT TIMING.TIMING for a timing diagram. (The diagram is for only one clock cycle, use multiple clock pulses in the same clock interval to use verify for more than one clock cycle.)

7. Use the ged WINDOW command to get a closer look.

8. A printout of the timing diagram results is in the file tvlst.dat

9. Use the ged hardcopy procedures to get a hardcopy of the timing diagram.

NOTE: You can set it up for various I/O studies using the case.dat file as discussed in chapter 6-73,75 of the scald manual. You will also find information concerning multiple clock pulses there.

SCALD NOTE 7.1

Timing Verifier

To get the scald machine's set of hold, setup, delay and etc. times in a pictorial format for a specific chip do the following in GED:

1. issue the ged command LIB TIME
2. issue the ged command EDIT *componentname*
3. use ged hardcopy procedures to get a hardcopy.

SCALD NOTE 7.2

Timing Verifier, page 1 of 2

More than one timing verifier run with changes:

1. Select the unix window and use the vi editor to edit the case.dat file. This file allows you to run several cases through the verifier changing signals, active clock interval periods and etcetera in each case.
2. Add to the file using the following format to cause changes.
   'signame' <15..0> = '1' , for example
   include the single quotes and keep the number of bits indicator (<15..0>) outside of the single quotes as shown.
   This example sets the bus to 1. verify does not simulate operations such as setting up an adder with S<3..0> = '1001' this is not a valid condition, set it to '1' or '0'
   Another example is: 'signame' <15..0> = '!S 14-17'
   This means that this signal is stable only during the clock intervals 14 thru 17.
3. To seperate cases use ; and insure you end the file with END.
   For example:

   ```
   'CLOCK' = '!C 10-15'
   ;
   'A' <15..0> = '!S 5-12'
   'CLOCK' = '!C 7-10'
   ;
   END.
   ```

4. When done editing the file, write it and then exit the vi editor.
5. If you've already compiled for time then issue the unix command **verify** and you should get two case runs for the example in step 3 above. If you have not compiled for time yet then do so first. The case.dat file is only used by verify and is not used by the compiler.
6. It there are no errors then issue the unix command **plottime**
7. For each case you will get a new timing.timing page.

SCALD NOTE 7.3

Timing Verifier, page 2 of 2

More than one timing verifier run with changes:

8. Note that you must use the exact signal name that is on your drawing. examples:

   CLK !C 14-17 use 'CLK !C 14-17'
   INPUT <15..0> use 'INPUT' <15..0> note the quotes!
   INPUT !S 8-12 <15..0> use 'INPUT !S 8-12' <15..0>

   the last example above sets this input stable only during the clock intervals 8-12. it is unstable the rest of the time.
   If you declare in your case file 'INPUT !S 8-12' <15..0> = '1'
   this means it is a stable '1' during the 8-12 interval.
   'INPUT !S 8-12' <15..0> = '!S 4-10,19' will override the original specification. This last specification means the signal is stable from the 4-10 interval and forever after the 19th interval.

SCALD NOTE 8.0

Scald Troubleshooting

Continuous Script File in Simulate:

This can be caused by using simcmd.dat as your script file. You must not use simcmd.dat directly as your script file. You must re-name the simcmd.dat file by using the **mv** or **cp** unix commands to something descriptive of your drawing name such as simlab1.dat. This condition can also be caused by not editing out bogus commands from your script file.

In order to get out of this continuous state, go to the unix window and issue the unix command **ps** then look at your current processes. Near the bottom of the list should be a process which indicates something to do with GED or SIM. Note the PID (process ID number) of that process. Issue the unix command **kill** <*PID*>

EOF Error in Simulate:

Occasionally you may get an error in simulate that shows up as an EOF (end of file) error. In order to get out of this state you should go to the unix window and issue the unix command **ps** then look at your current processes. Near the bottom of the list should be a process which indicates something to do with GED or SIM. Note the PID (process ID number) of that process. Issue the unix command **kill** *PID*

Scald Terminal is locked up:

If the terminal doesn't seem to respond to any input (locked up), go to the maintenance terminal and see if it is locked up. If it is not locked up, login to your account and issue the unix command ps then look at your current processes. Near the bottom of the list should be a process which indicates something to do with GED or SIM. Note the PID (process ID number) of that process. Issue the unix command **kill** <*PID*>

If both terminals are locked up, then press the WHITE button on the rear

of the scald and follow the instructions subsequently given on the display screen. You will be instructed to give the date in a certain format and you should answer any questions with y followed by a carriage return. If the screen goes blank and sort of looks like it has lost "sync" for a lengthy period of time then press the white button again. etc.

SCALD NOTE 8.1

Scald Troubleshooting

Unable to Obtain a Hardcopy:

If you are unable to obtain a hardcopy following the hardcopy procedures, press the pause, form feed and advance buttons on the versatec printer simultaneously and release them. Wait for the paper to feed and a test pattern to be produced. If there is a good test pattern, then the printer is alright. Typical printer problems include not having the paper properly installed or simply powering down and powering up the printer will clear the problem.

If you are on scald 2, have scald 1 try to make a hardcopy. If scald 1 can not make a hardcopy either, then the problem is most likely with scald 1. If scald 1 is able to make a hardcopy then scald 2 is the problem and you should then be sure that all of your drawings are written and logout. Then press the WHITE button on the back of scald 2 and follow the instructions subsequently given on the display screen. You will be instructed to give the date in a certain format and you should answer any questions with y followed by a carriage return. If the screen goes blank and sort of looks like it has lost "sync" for a lengthy period of time then press the white button again, etc.

If scald 1 is unable to print then the problem is clearly with scald 1. The user of scald 1 should be sure that all drawings are written and logout. Then press the WHITE button on the back of scald 1 and follow the instructions subsequently given on the display screen. You will be instructed to give the date in a certain format and you should answer any questions with y followed by a carriage return. If the screen goes blank and sort of looks like it has lost "sync" for a lengthy period of time then press the white button again, etc.

No Swap Space Error:

If you get an error which indicates no swap space. press the WHITE button on the rear of the scald and follow the instructions subsequently given on the display screen. You will be instructed to give the date in a certain format and you should answer any questions with y followed by a carriage return. If the screen goes blank and sort of looks like it has lost "sync" for a lengthy period of time then press the white button again, etc.

SCALD NOTE 9.0

GED Notes

Show Net:

In ged issue the command SHOW NET from the keyboard or by picking it from the ged menu with the cursor if it is on the menu. Use the cursor and select unlabelled signal lines with the yellow button. The strange looking names for these lines are referenced by error messages. the simulator and etc. In simulate you can go back to the drawing and open any signal line and it will be referenced by this strange looking name if you have not given the signal line a signame. You can then monitor this signal line in the simulator as you would a signal line that you have already given a signame.

Open Collector Implementation:

1. Be sure that the phantom library is being referenced in your startup.ged and compiler.cmd files.
2. Change the maximum delay property to a reasonable value. This means that you should issue the ged command EDIT PHAN 2 AND for example to edit the library 2 input phantom and gate and change the maximum delay property on the drawing to a lower more reasonable value, if necessary. in accordance with a value from a data book for instance. Then issue the ged command WRITE . It will stay the value you set in the library only for the remainder of your terminal session.
3. Issue the ged command ADD PHAN 2 AND to add a 2 input phantom and gate to your drawing for example and wire it up as you would any other component.
4. No resistor is needed.
5. See scald reference manual volume 2 chapter 11 (component libraries) for a description of phantom bodies.

SCALD NOTE 9.1

Merges

Merges:

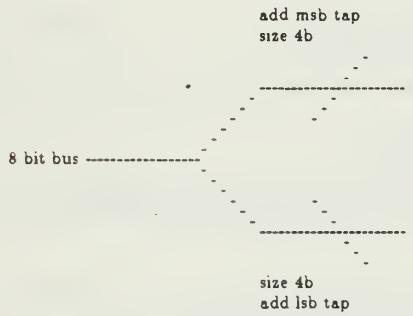
See scald reference manual volume 2 chapter 11 (component libraries) for a description of merges. taps, etc. Merges are used primarily for buses. For example they enable you to use 1 bus line instead of 4 separate lines and represent the 4 bits (or 4 bytes) on the line as a 1 digit hexadecimal number (or 4 digit hexidecimal number). Notice when you change the version of a component that you have added to your drawing (for instance a 74LS181 ALU), that one version has 4 discrete input and output lines and the other version of the same component has only 1 input and output line. The later version is for use in a bus and represents 4 lines and its current value is represented in simulate by a hexidecimal number. You could use a 4 merge to break out the bus with the MSB being on top unless specified or merge 4 lines together into a bus. Merges can be placed anywhere on a drawing and do not have to be physically connected as long as the signames are explicitly given and match corresponding signames elsewhere on the drawing.

Note below one interesting application of a merge. To get this on your drawing issue the ged command ADD 4 MERGE and then issue the ged command SIGNAME to name the lines. In this example the signals Q0(H) to Q3(H) refer to the outputs of D flip-flops elsewhere on the drawing. This merge allows one to use the bus Q<3..0> in the simulator to represent as a 1 digit hexidecimal number what would have been 4 seperate 1 bit signal lines.

SCALD NOTE 9.2
GED Notes

Taps:

Taps can be used to insert or extract a line or lines on a bus and can also be used just like a merge as illustrated below. Don't rotate taps: use the ged command VERSION or MIRROR . See scald reference manual volume 2 chapter 11 (component libraries) for a description of merges. taps. etc. In the illustration below 2 taps are used to breakout a bus much like a 2 merge. Assuming the 8 bit bus exists on your drawing. issue the ged command ADD MSB TAP then use the ged command VERSION or MIRROR to get the proper orientation and then make connections. Next issue the ged command ADD LSB TAP and etc. Then give the taps the property of size 4b.



Body Glitch:

Always put a wire on a pin in a logic drawing before attaching a signal name if it is a body you created. Otherwise when you get to the point of ignore group.wrk use group.wrk when it restores your body there was no wire for the signal to be attached to and the ignore group.wrk deleted your body with the logic drawing signals attached. Moral - the body comes back - the signal names don't!

Body note:

1. Don't use a page.
2. Keep the wire input and output stubs (pins) short and when subsequently connecting to them do not cross another stub (pin) unless you want to connect to it. Wires can cross wires - not stubs (pins) !

SCALD NOTE 9.3

GED notes

To insure the signame attaches to the wire:

1. Issue the ged command SIGNAME
2. Move the cursor over the wire you desire and press the yellow button. a * will appear on the wire.
3. Issue the ged command NAME
4. issue the ged command MOVE and move the name anywhere you want it. it is still connected to that wire.

Size property (bit size of a component):

1. Issue the ged command PROPERTY
2. Move the cursor over the component and press the yellow button.
3. Type in the ged command SIZE 8b for example. In this case 8 bits were chosen.
4. The component is now effectively multiplied by the size and signals input to and output from the component must be of the right size. For this example an 8 bit signal might be A<15..8>. If you don't get the match right you will get a PATHSIZE MISMATCH ERROR.

To change letter size of notes:

1. Issue the ged command SET SIZE .5 for example.
   This will set the size of the notes to .5 of the default note size.
   You may use any fraction of the default size.

To set constant value signals (hardwire values):

1. Make the assigned signame 16 # FFFF . for example.
   In this example 16 is the number of bits, # is the code for a constant value, and FFFF is the constant value.
   Other examples are: 2 # 01 , 4 # 7 , 8 # FF , 16 # 01AB or for a binary signal just the number 0 or 1 is acceptable.
2. Two or more signals can have the same constant value signame.

25 Apr 1986 L. Weist D. Schaeffer

SCALD NOTE 9.4

GED notes

Use of puck buttons:

YELLOW - starts and stops a line (wire) at a grid intersection (or wire intersection).

BLUE - starts and stops a line (wire) at a vertex, which is the end of a pin or the end of a wire. When placing dots at wire intersections the blue button is much easier to use because it sets the dot to the nearest intersection, therefore, you don't have to be right on top of the intersection.

GREEN - changes the wire bend direction in sequence from upper right bend to lower left bend to straight line between points.

WHITE - used for group moves. It is used to grab a group you have made instead of a chip or wire (use the yellow button for chip and wire moves).

Use the blue button to connect pin to pin.
Use the blue button then the yellow button to connect pin to an intersection.

SCALD NOTE 10.0

Creating a Body, page 1 of 2

Creating a body is one of the most powerful aspects of hierarchical computer aided design. When you have finished a drawing you can make the entire drawing into a component (body) that can be added to other drawings. You are making your own "logic chip" out of your drawing. For example. remember that a 74ls181 ALU chip is simply a lot of logic gates "wired" together in the silicon chip. You can see that the logic heirarchy does not exactly start with a 74ls series logic chip that you add to your drawing.

In order to make your drawing into a body, certain changes must be made to your drawing. Because of these changes that must be made to your original "stand alone" drawing, you may want to first make a copy of your original drawing with a new name before making the necessary changes. To do this, in ged issue the ged command EDIT <*drawingname*> then when your drawing is displayed issue the ged command WRITE <*newdrawingname*>. This creates a copy of your original drawing called <*newdrawingname*>. Then proceed with the below steps:

1. Change all signal names to be used as inputs to and outputs from your drawing (body) to include an \I on the end of the signame. For example A<15..0>\I.

2. If your drawing included a clock, change the clock signame to CLK\I (delete the pulse times from the clock signame).

3. When complete with the signame changes write the drawing.

4. Issue the ged command EDIT <*drawingname*>

5. You will notice an X and your drawing name on the screen.

6. Issue the ged command SPLIT and pick the drawing name on the ged screen with the yellow button. This will allow you to move the drawing name without moving the X. Do not move the X.

SCALD NOTE 10.1

Creating a Body. page 2 of 2

7. Do not add a page. Drawing title and abbreviation is not required.

8. Issue the ged command GRID .05 1 (be sure that the body grid size corresponds to your drawing grid size!)

9. If your body will resemble a known logic symbol, issue the ged command ADD *symbol* to add the symbol to the drawing. Center the symbol around the X and then issue the ged command SMASH then start deleting all unnecessary features and text. Leave the X alone and move the drawing name within the symbol as necessary. You can also use the ged command WIRE to draw your own symbol shape.

10. Issue the ged command WIRE to add short wires to where you want input and output connections to your symbol (if not already present). Avoid adding short wires to inversion circles if present. Then issue the ged command DOT to dot the ends of all wires and inversion circles.

11. Issue the ged command SIGNAME to name the wires. The names must be the same as the drawing names except do not include the \I, they must be outside of the symbol body and be sure that they are attached to the proper wire.

12. Issue the ged command NOTE and add identifying pin signal names (if not already present) to the inside of the symbol (body). Only these notes will show when your body is added to a drawing. You can change the note lettering size to a decimal fraction of the current size by issuing the ged command SET SIZE .5 for example..

13. Write the body drawing. You now have a "chip" called *drawing name*

SCALD NOTE 11.0

Plottime

For more detailed timing plot:

1. After simulation is complete select or type the simulate command PLOT
2. select the unix window and issue the unix command **plottime**
3. issue the simulate command EXIT to get back to just the ged window.
4. issue the ged command EDIT TIMING.TIMING
5. press RF1 to get a hardcopy.
6. if there is more than 1 page of timing then next issue the ged command EDIT TIMING.TIMING.1.2 for the second page. then get a hardcopy, and etcetra.

To get up to 50 timing lines on a plot:

1. select the unix window and use vi to edit the td.cmd file
2. change the SIGNALS_PER_PAGE value to what you want.

To get more than 1 plottime plot during the same login period:

1. when your simulation run is complete issue the simulate command
   PLOT <*filename*>
   filename is whatever you want to call the plot.
2. select the unix window and use vi to edit the td.cmd file,
   insert INPUT '<*filename*>'; below the directory line,
   then insert OUTPUT '<*filename*>.*timing*';
   write the file, exit the vi editor and issue the unix command **plottime**
3. Select the ged window and issue the ged command EDIT <*filename*>.*timing*
4. now get a hardcopy of the plot by pressing RF1.

20 Mar 1986 D. Schaeffer

SCALD NOTE 12.0

Simulator Update

To use the added capability and convenience of the recently updated fullscreen simulator.

1. Initially login to unix using the full screen.

2. Issue the unix command **ged** to get into ged.

3. Issue the ged command EDIT <*drawingname*> to make the appropriate drawing available for editing. This is optional, but recommended.

4. Issue the ged command UNIX to get into full screen unix again.

5. Issue the unix command **simulate** <*drawingname*>

6. You will be placed in full screen simulate after a significant wait for the compiler to run. You don't have a ged window to pick signames from or to even look at so it would be wise to have a hardcopy or script file already prepared before a fullscreen simulate.

7. As of yet no one has found out how to get a direct hardcopy from fullscreen simulate. In order to get a hardcopy issue the command PLOT in the full screen simulator and follow the procedures for plottime.

SCALD NOTE 12.1

Simulate Notes

Simulate data entry and script files:

There are alternate ways of entering data for your simulation. You can either enter data by way of actual keyboard entry at the time of simulation or by editing a script file beforehand. Whichever method is used one can use abbreviated simulate commands. Below are 3 columns of commands that are equivalent.

commands #1	commands #2	commands #3
WAVE 0 2000	wav 0 2000	w 0 2000
HISTORY 5000	his 5000	h 5000
OPEN A	o A	o A
OPEN B	o B	o B
OPEN C	o C	o C
OPEN A	o A	
DEPOSIT 1	d 1	d A,1
OPEN B	o B	
DEPOSIT 0	d 0	d B,0
SIM 100	sim 100	s 100

SCALD NOTE 13.0

Loading Memory Devices

To use a PROM or EPROM the device must be selected from the MEMORY library (scald reference manual pg 11-19,96). Loading a RAM may be performed in a similar manner to the one outlined below but has not been tried.

The device is wired up in ged just like any other component and compiled for simulate just like any other circuit.

Before proceeding further read pages 7-32 and 7-46,47 of the scald reference manual.

The ROM/EPROM will be loaded from a unix file that you edit while in the unix window. Page 7-46 of the scald reference manual shows an example of the file. An example called memory1.dat follows:

memory1.dat:

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 7 .. 0;
MEM_BLOCK 0.4;
  0000 0000;
  0000 0001;
  0000 0010;
  0000 0111;
END_MEM_BLOCK;
END.
```

The remainder of the memory load is performed in the ged simulator.

However, you must have path information from the cmplst.dat file. Each device in scald will be given a path number such as 1P or 2P. In the cmplst.dat file after the STARTING TO PROCESS DRAWINGS is a list of paths and pathnames. These will be needed in the simulator (when in ged). It might be a good idea to get a hardcopy of the cmplst.dat file to make it easier to reproduce the pathnames. An example of the pertinent area of the complst.dat (output from simulate compile) follows:

```
Starting to process drawings
```

```
(MEMTEST .27LS19.2P)
(MEMTEST .27LS19.2P MEM3P)      - pathname for 2nd 27LS19

(MEMTEST .27LS19.2P TSB2P)

(MEMTEST .27LS19.2P BUF4P)
(MEMTEST .27LS19.1P)
(MEMTEST .27LS19.1P MEM3P)      - pathname for 1st 27LS19

(MEMTEST .27LS19.1P TSB2P)

(MEMTEST .27LS19.1P BUF4P)
```

Now, select the ged window and issue the SIMULATE command. then when in the simulator issue the simulate commands PLOT 0 1000 and HIS 5000. To load the memory device (burn the EPROM/ROM/PROM) issue the simulate command MEML or MEMLOAD. You will then be prompted for information.

For example, if you are loading the PROM with 1p in the pathname. in simulate. type in the full pathname followed by a carriage return. In this example MEMTEST.27LS19.1P MEM3P is the full pathname. MEMTEST is the drawing name, 27LS19 is the component (PROM), and 1P differentiates this component from other 27LS19 components.

Then issue the simulate command MEMLOAD or MEML. Once again you will be prompted for information.

Now type in the name of your file that you made for loading the device followed by a carriage return. In our example, the name of the file is memory1.dat. You have now loaded the the 1P PROM with the data from the file memory1.dat.

You can type MEMLOAD again and type in the second pathname to load more memory components. You can also type NEXTMEMORY, the scald will find the next pathname and will give you the pathname. Find the register then type MEMLOAD and then type in the file you want loaded.

Then OPEN, DEPOSIT and SIMULATE the hours away. You can use a simcmd.dat file like the example that follows:

simmem.dat:

```
history 5000
wave 0 1000
mempath
(memtest .27ls10.1p mem3p)
memload
memory1.dat
nextmemory
memload
memory2.dat

OPEN Q<15..0>

OPEN CS*\I
DEPOSIT 0

OPEN A<4..0>\I
DEPOSIT 0
SIMULATE 100
```

VIII. APPENDIX B

```
*******************************************************************************
*******************************************************************************

# >>>>>> FILENAME DOIT <<<<<<<<<<<<<<<<<<<<<<<
# This is the compiler exec file.  It contains
#  calls to all the subroutines filed in different
#  directories.

cd FILENAMECHANGE
Fnchange
cd ..
echo -n "Do you wish to continue ? (y or n): default is no "
set a = $<
if ($a != y)  then
        goto end
endif
echo "running First in directory FIRST"
cd FIRST
First
cd ..
cd THESISPROP
echo "running Assign in directory THESISPROP"
Assign
cd ..
cd SIGSORT
echo "running Sigsort in directory SIGSORT"
Sigsort
cd ..
cd FINALSORT
echo "running Finalsort in directory FINALSORT "
Finalsort
cd ..
cd THESISDEF
Thesisdef
cd ..
end:

*******************************************************************************
*******************************************************************************
```

```
# >>>>>> FILENAME Fnchange <<<<<<<<<<<<<<<<<<<<
# ---------- directory FILENAMECHANGE ------
# **********input file name user file *********
# ********** output file name td **************
# This routine is a sub command file that creates a
#  duplicate of the original connectivity file for
#  the compiler's use.  The original file is left
#  unaltered.  It then checks for obvious errors
#  in the file and provides a screen printout and a
#  separate file called error.dat for later printout
#  if desired.


loop:

# The next 5 lines get the file name interactively from
#  the user.

  echo -n "Please enter the input file name" :
  set a = $<
  echo "ARE YOU SURE "$a "IS THE CORRECT FILE ? (y or n); "
  set b = $<
  if ($b != "y") then
                  goto loop
  endif
echo "file copied to file name td for compiler use"

# The next 3 lines copy this new file named td to the
#  necessary directories for later use.

cp ../../$a td
cp ../../$a ../FIRST/td
cp ../../$a ../THESISDEF/td
echo "Running an error check on the input file"

# This awk routine call checks the input file for obvious errors.

awk -f Noconnectck.cmd td

# This awk call sends the errors to the screen with comments.
```

```
awk -f Errorout.cmd error.dat

= Finally the error file is moved to the main directory for
#  user printout if desired, and the td file is deleted to
#  save space in the system.

mv error.dat ../../error.dat
rm td

*******************************************************************************
*******************************************************************************
```

```
# >>>>>>> filename Noconnectck.cmd <<<<<<<<<<
# ---------- directory FILENAMECHANGE ----+-------
#  ********** input file td ***********
# *********  output to screen and file error.dat ****
# This routine checks for obvious errors in the input file
#  and outputs an error locating the source of the problem.
#  The Doit routine asks if the user wishes to continue because
#  I have not found a method of making the main program stop
#  based on an output from a subprogram.
# Set the field separator as and " symbol and create the error.dat
#  file printing a title in the file.  Then initialize the three
#  variables.  These will be used to detect whether or not a start
#  state, any signal definitions or title exists.

BEGIN{ FS = "\"": file = "error.dat":
     print "error.dat file" > file
        strtstatexists = 0
        signamexists = 0
        titlexists = 0}
# Search all input lines and get the property number (the number in field
#  10 with the P attached to it) if it exists.  Place it in the variable
#  named errpropnum.

$10 ~ /[0-9.P]/{errpropnum = $10}

# Check the signal number field ( field 3) of the following "pin name"
#  lines for a zero: in. out, setq, zcond, yes, no, a, process,
#  zgoto, outstart. outother. and title.  The /^out$/ denotes that the
#  search will be conducted for exactly the text within the ^ and $.
#  A zero indicates that this pin name is not connected to any other
#  pin or signal name in the circuit.  Print out an error and include
#  the property number of the symbol involved (stored in the errpropnum
#  variable).

$3 ~ /^0;$/ && $2 ~ /^out$/{
 print "The intrastate symbol with property number " errpropnum > file
          print " has an unconnected 'out' pin" > file}
$3 ~ /^0;$/ && $2 ~ /^in$/{
 print "The state or intrastate symbol with property number " errpropnum > file
          print " has an unconnected 'in' pin" > file}
$3 ~ /^0;$/ && $2 ~ /^setq$/{
```

```
print "The intrastate symb  with property number " errpropnum  file
          print " has no function assigned." > file}
$3 ~  ^0;$ && $2 ~  ^zcond$ {
        print "The conditional symbol with property number " errpropnum > file
          print " has no function assigned." > file}
$3 ~ /^0;$/ && $2 ~ /^yes$/{
        print "The conditional symbol with property number " errpropnum > file
          print " has an unconnected 'yes' pin" > file}
$3 ~ /^0;$/ && $2 ~ /^no$/{
        print "The conditional symbol with property number " errpropnum > file
          print " has an unconnected 'no' pin" > file}
$3 ~ /^0;$/ && $2 ~ /^a$/{
        print "The conditional symbol with property number " errpropnum > file
          print " has an unconnected 'a' pin" > file}
$3 ~ /^0;$/ && $2 ~ /^process$/{
        print "The process section of the title block " > file
          print " is undefined" > file}
$3 ~ /^0;$/ && $2 ~ /^title$/{
        print "The title section of the title block " > file
          print " is undefined" > file
                titlexists = 1}
# Check that a title block exists and print an error at the end if it
#  doesn't.  The variable titlexists is used to designate this.

$2 ~ /^title$/{titlexists = 1}
$3 ~ /^0;$/ && $2 ~ /^zgoto$/{
        print "The state symbol with property number " errpropnum > file
          print " has no function assigned." > file}
$3 ~ /^0;$/ && ($2 ~ /^outstart$/ || $2 ~ /^outother$/){
        print "The state symbol with property number " errpropnum > file
          print " has an unconnected 'out' pin" > file
                strtstatexists = 1}
# Check for the 2 symbols STARTSTATE and SIGNALNAMES.  If they aren't
#  on the drawing set the flags for later error printout.

$2 ~ /^outstart$/{strtstatexists = 1}
$2 ~ /^def$/{signamexists = 1}
```

```
= Print out errors based on the 3 flags after the entire input file
=  has been read.

END{ if(strtstatexists == 0) {print "There is no start state" > file}
   if(titlexists == 0) {print "There is no title block" > file}
        if(signamexists == 0) {print "There is no signalnames block" > file}}

*******************************************************************************
*******************************************************************************
```

```
# >>>>>>>>>>>>filename Errorout.cmd <<<<<<<<<<<<
# ++++++++++ directory FILENAMECHANGE ++++++++++
# ***********input file is error.dat ***********
# ***********output file is the screen *********
# This routine checks the error output file for more than
# one line (ie an error has been found) and sends the output
# of the error.dat file to the screen.  If no error is found
# the statement "There were no interconnect errors detected..." is
# sent to the screen.

# Send the initial message to the screen.

BEGIN{print " "
    print "ERROR CHECK RESULTS ARE SAVED IN ERROR.DAT"
    print " "}

# NR is the number of records in the file (updated after each
#  line of input code is read).  Print $0 prints the entire line
#  to the output file (in this case the screen) as it appears in
#  the input file.  Here it is printing the detected errors if
#  more lines than just the title exist in the error file.

{if (NR > 1){ print $0
                flag = 1}}

#  If there were no errors then send the message below.

END{if(flag == 0) {print "THERE WERE NO INTERCONNECT ERRORS DETECTED"
                   print "IN THE INPUT FILE."
            print " "}}

*******************************************************************************
*******************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>> COMMAND FILE NAME First <<<<<<<<<<
# ---------- directory FIRST ----------
# ********** initial input file td **********
# **********  final output file thesis.dat *****

echo "converting the connectivity file"

# This routine removes the "z" from the signals
#  "zcond" and "zgoto" in the td file (the copy of
#  the original connectivity file from the scald system).
#  This allows property assignment routines
#  to key on the conditional lines and act on the
#  appropriate lines of code located below the cond
#  line in the modified connectivity data file
#  (now called thesis.dat).  The "z" was added to the
#  cond and goto pin names in the SCALD DRAWINGS to
#  force the SCALD generated CONNECTIVITY FILE to list
#  these items first in their property block or group.
#  Note that in this connectivity file pin names are listed
#  in reverse alphabetical order (hence the "z").

# Run the awk routine that removes the "z's".

awk -f changezcondzgoto.cmd td
echo "sending "thesis.dat" to THESISPROP."

# Copy and (then remove) the newly generated thesis.dat file
#  to the two directories that need it.  Remove the td file
#  to save space.

mv thesis.dat ../THESISPROP/thesis.dat
rm td

**********************************************************************
**********************************************************************
```

```
**********
**********

= >>>>>>>> file name changezcondzgoto.cmd <<<<<<<<<
# ++++++++++ directory FIRST +++++++++++++++++++++
# ***** USE td   AS INPUT FILE **********
# ******uses thesis.dat   as output file **********
# This segment changes the "zcond" and "zgoto" lines
#  by removing the z's and duplicating the input file
#  in all other respects.  The "z" was added to the
#  signal name on the scald system to ensure it appeared
#  at the top of its property section file in the
#  connectivity file output.  This is necessary for later
#  processing in this conversion program.  Note that the
#  signal names appear in reverse alphabetic order in the
#  connectivity file.

# Set the input and output field separator to the " symbol
#  and create the output file thesis.dat with no title.

BEGIN {FS = "\""; OFS = "\"";file = "thesis.dat"}

# Search for zcond or zgoto in the 2nd field of the input
#  file lines and print the line with the z removed.  Otherwise
#  print the line as it appears in the input file.

{
if($2 ~ /^zcond$/){
                print "\"cond", $3 > file
                }
else if($2 ~ /^zgoto$/){
                print "\"goto", $3 > file
                }
else {print $0 > file}
}


**********
**********
```

```
*****************************************************************************
*****************************************************************************

=
= >>>>>>>>>> COMMAND FILE NAME Assign <<<<<<<<<<<<<<
= ---------- directory THESISPROP ++++++++++++++
# This command file calls 6 awk routines that are
# individually documented in detail.  Basically this
# section reduces the original thesis.dat file into
# a more compact file.  This compactness is a result
# of combining "pin name" lines and combinations of their
# corresponding "signal names" based on the property numbers
# and signal numbers.  The directory FINALSORT requires that
# any "a" or "in" pin name line appear before all other lines
# when sorted by signal number.  The "a" and "in" pin name lines
# are therefore pulled and inserted at the top of the final file
# to allow the next directory (SIGSORT) to sort the file in this
# manner. The results are found in file property4.dat.  Note that
# all created data files are removed by the last line of this file.
# To see these files after a run comment out the "rm" line as
# desired.


echo "Sorting and assigning properties (see Assign)"
awk -f Propassign.cmd thesis.dat
awk -f Setqcond.cmd property.dat
awk -f Inassign.cmd property2.dat
awk -f Outassign.cmd property3.dat
awk -f Condassign.cmd property5.dat
awk -f alinepull.cmd property4.dat
cat apull1.dat apull2.dat > property4.dat
echo "sending property4.dat file to SIGSORT directory"
cp property4.dat ../SIGSORT/property4.dat
rm *.dat


*****************************************************************************
*****************************************************************************
```

```
*************************************************************************
*************************************************************************

= >>>>>>>>>>> filename Propassign.cmd <<<<<<<<<<<
# ---------- directory name THESISPROP ------------
# ****** USE THESIS.DAT AS INPUT FILE **********
# ******uses property.dat as output file *********

# this segment  pulls the property number
# off of the property line and attachs it to each line
# of code in that property, then it deletes all un-needed lines.

# Set the input and output field separators to " and the input
#  record separator to ;, then create the property.dat file
#  and put a title in it.

BEGIN {FS = "\""; RS = ";"; OFS = "\""; file = "property.dat";
     print ">>>>>>>> PROPERTY.DAT FILE <<<<<<<<;" > file}

# If it is a signal line (at the beginning of the file) then
#  print it out to the file.

$2 ~ /[0-9]/{print $2, $3,";" >file}

# Find the lines with the property number in field 14 (the number
#  with the P in it) and store the number in the variable prop.

$2 ~ /^%$/ && $14 ~ /[0-9.P]/{prop = $14}

# Search for all lines between property number lines that have
#  signal numbers in the 4th field and are not "def" lines and
#  add the property number at the beginning of the line.  Print
#  it in the file. Note in this case the previous carrage return
#  is in field number 1 since the record separator is ;.

$4 ~ /[0-9]/ && $3 !~ /^def$/{print prop, $3, $4, ";" >file}
END {print "END." > file}


*************************************************************************
*************************************************************************
```

```
# >>>>>>>> filename Setqcond.cmd <<<<<<<<<<<<<<
# -------- directory THESISPROP ----------------
# ***** USE PROPERTY.DAT AS INPUT FILE ***********
# ******uses property2.dat as output file **********
# This segment pulls the signal names and attaches
#  them as required to the "cond", "goto" and "setq" lines.
#  This sets up the format for lisp code for the
#  conditionals and setq's.  NOTE THAT A SIGNAL TEST WITHIN A
#  CONDITIONAL DOESN'T REQUIRE PARENTHESIS.HOWEVER, A FUNCTION
#  SUCH AS "IS A = B ?" REQUIRES PARENTHESIS AROUND THE FUNCTION.
#      ex. (= a b) is an equality test for two integers.
#  THIS REQUIREMENT IS MOST EASILY MET AT THE TIME THE DRAWING IS
#  MADE ON THE SCALD SYSTEM....There is room for inprovement here.

# Set the field and record separators and create the property2.dat
#  file with a title line.

BEGIN {FS = "\"": RS = ";": OFS = "\"";file = "property2.dat"
     print ">>>>>>>> PROPERTY2.DAT FILE <<<<<<<<" > file}

# Load 2 arrays, one with the signal number and one with the
#  signal name using the signal number as the array index.

$2 ~ /[0-9]/ && $2 !~ /P/{y[$2] = $2
          x[$2] = $3}

# If the 4th field contains a valid signal # (held in the array)
#  then check the third field for a setq, goto, or cond and add
#  the corresponding signal name to that line.  Otherwise just
#  print out the line.
{
        if (($3 ~ /^setq$/)&& ($4 == y[$4])){
                        print $2, "(" $3 " " x[$4] ")", ";" >file
                }
   else if (($3 ~ /^goto$/)&& ($4 == y[$4])){
                        print $2, "(go " x[$4] ")", $4,  ";" >file
                }
   else if (($3 ~ /^cond$/)&& ($4 == y[$4])){
                        print $2, "(" $3 " (" x[$4], ";" > file
                        }
  else{print $2, $3, $4, ";" > file}
```

```
}

= NOTE THAT THIS WORKS BECAUSE THE SIGNAL NUMBERS ANd NAMES ARE LOCATED
#  AHEAD OF ANY LINE TO BE MODIFIED IN THE INPUT DATA FILE.
#  THEREFORE, THE ARRAY CAN BE LOADED AND USED LATER IN THE PROGRAM
#  TO MANIPULATE DATA LOCATED IN THE DATA FILE AFTER THE ARRAY IS LOADED..

*************************************************************************
*************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>>> filename lnassign.cmd <<<<<<<<<<<<<<<<
# ++++++++++ directory THESISPROP ++++++++++++++++++
# ****** USE PROPERTY2.DAT AS INPUT FILE **********
# ******uses property3.dat as output file *********
# This segment pulls the setq or goto definition for a property
# off of the setq or goto property line and attachs it to the line
# of code that contains the word "in" within the same property
# group.

# Set the field and record separators, create the file property3.dat
#  and add a title line to it.

BEGIN {FS = "\""; RS = ";"; OFS = "\""; file = "property3.dat";
     print ">>>>>>>>> PROPERTY3.DAT FILE <<<<<<;" > file}

# Search for lines with "(setq " or "(go " (including the space
#  after the word) and if found print out the line unchanged and
#  load the signal name ( located in field 3) into the variable
#  called prop.  This specific search hopefully prevents any
#  pickup of a user designated signal name that might contain
#  somewhere in his signal name the pattern "in" or "setq".  An
#  example is "sigin".  If the search had been just for the pattern
#  "in" this line would have tripped the search.
# The command "next" at the end of the line causes
#  the program to go to the next line in the input file.  This
#  prevents the print action at the end of this file from double
#  printing the line.

$3 ~ /\(setq /{prop = $3
        print $2. $3, $4, ";" > file
                        next}
# Note the ")" added to the variable string in the variable prop.
#  during the "go" search below.
#  This allows the program to close off the end of a conditional
#  action with the go action, thereby reducing the parenthesis
#  matching problems of the completed MACPITTS input code.

$3 ~ /\(go /{prop = $3")"
        print $2, $3, $4, ";" > file
                        next}
```

```
= If the 3rd field of the input line contains the word "in" then
=  insert the current signal name contained in the variable prop
=  and print out the new line to the file.

$3 ~ /^in$/{print $2, $3, prop, $4, ";" >file
                        next}  .

# Print any line not already printed.

{print $2, $3, $4, ";" > file}
```

```
*******************************************************************************
*******************************************************************************
```

```
*****************************************************************************
*****************************************************************************

= >>>>>>>>>>>> filename Outassign.cmd <<<<<<<<<<<<<<<
= ++++++++++ directory THESISPROP ++++++++++++++++++++
# ****** USE PROPERTY3.DAT AS INPUT FILE **********
# ******uses property5.dat as output file *********
# This segment pulls the state label and attaches it to the
# "outstart" and "outother" lines.

# Set the field and record separators and create the property5.dat
#  file with a title line.

BEGIN {FS = "\""; RS = ";"; OFS ="\""; file = "property5.dat";
     print ">>>>>>>>> PROPERTY5.DAT FILE <<<<<<<<;" > file}

# Load the 2 arrays. x is the signal name and y is the signal number.
#  Both are indexed on the signal number.

$2 ~ /[0-9]/ && $2 !~ /P/{y[$2] = $2
            x[$2] = $3}

# If a line with "(go " (including the space after go)
#  is found in field 3 then check the 4th field for a signal
#  number match found in the y array.  If found, assign the
#  variable prop the corresponding signal name.  The "\" before
#  the "(go " is a required delimiter to allow the "(" to be
#  checked as a text symbol.

$3 ~ /\(go /{if($4 == y[$4]){prop = x[$4]}}

# If an "outstart" or "outother" line is found (field 3) then
#  print it in the file adding the previously assigned signal
#  name located in variable prop.  If not found then print out
#  any line except the "(go " line (no longer needed).

{if ($3 ~ /^outstart$/ || $3 ~ /^outother$/)
                          {print $2, $3, prop, $4, ";" >file}
else if($3 !~ /\(go /){print $2, $3, $4, $5  ";" > file}
}
```

```
= The first block says if the 3rd field contains
= the word "in" then print out a new line containing the original line data.
= the setq property (prop), and the 2 fields of interest ( 2 and 3) in a
# new file .  If the line is not changed then print it as
# it appears in the input data file.

*******************************************************************************
*******************************************************************************
```

```
*****************************************************************************
*****************************************************************************

= >>>>>>>>>> filename Condassign.cmd <<<<<<<<<<<<<
= ---------- directory THESISPROP ----------
# ****** USE PROPERTY5.DAT AS INPUT FILE **********
# ******uses property4.dat as output file *********
# This segment pulls the cond definition for a property
# off of the setq or goto property line and attachs it to the line
# of code that contains the word "in" within the same property
# group.

# Set the field and record separators, create the file property3.dat
#  and add a title line to it.

BEGIN {FS = "\""; RS = ";"; OFS = "\"": file = "property4.dat";
      print ">>>>>>>>> PROPERTY4.DAT FILE <<<<<<;" > file}

# Search for lines with "(cond " (including the space
#  after the word) and if found print out the line unchanged and
#  load the signal name ( located in field 3) into the variable
#  called prop.  This specific search hopefully prevents any
#  pickup of a user designated signal name that might contain
#  somewhere in his signal name the pattern "in" or "setq".  An
#  example is "sigin".  If the search had been just for the pattern
#  "in" this line would have tripped the search.
# The command "next" at the end of the line causes
#  the program to go to the next line in the input file.  This
#  prevents the print action at the end of this file from double
#  printing the line.

$3 ~ /\(cond /{prop = $3
         print $2, $3, $4, ";" > file
                          next}

# If the 3rd field of the input line contains the word "yes" then
#  insert the current signal name contained in the variable prop
#  and print out the new line to the file.

$3 ~ /^yes$/{print $2. $3, prop, $4, ";" >file
                          next}

# Print any line not already printed.  Note that $5 is included because
#  this field exists on the outstart and outother lines and must be
```

```
=  included in the reprint of those lines.

{print $2, $3, $4,$5, ":" > file}

# This is the same code as the Inassign.cmd file, but it searches for
#  the cond and yes lines.

*******************************************************************************
*******************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>> FILENAME alinepull.cmd <<<<<<<<<<
# ---------- directory THESISPROP ++++++++++++++
# ******** input file property4.dat **********
# output files apull1.dat and apull2.dat *******
# This progam assures that any "a" or "in" line occurs before
# any other signal matched lines in the sigsort routines.
# This is required to get the correct results in the
# Finalsort routines.  These routines key on the "a" or "in"
# lines and then perform modifications on the code later in the
# input file based on the "a" and "in" lines.

# Set the field separators and create 2 output files.  Apull1.dat
#  contains any "a" or "in" lines and apull2.dat contains all others.
#  These will be concatenated later.

BEGIN{FS = "\""; OFS = "\""; file = "apull1.dat"; file1 = "apull2.dat";
     print ">>>>>>>>>>> APULL1.DAT FILE <<<<<<<<<<;" > file;
     print ">>>>>>>>>>> APULL2.DAT FILE <<<<<<<<<<;" > file1}

# Put "a" and "in" lines in file, all others in file1

{if($2 == "a" || $2 == "in"){print $0 > file}
else{print $0 > file1}}

**********************************************************************
**********************************************************************
```

```
# >>>>>>>> COMMAND filename Sigsort <<<<<<<<<<
# ++++++++ directory SIGSORT ++++++++++++++++
# ********** initial input file Property4.dat ****
# ********** final output file sortedsig *********
# The numberofsig.cmd file determines the number of
#  signals in the file input and limits the sort to that
#  number thereby saving time.  This number is placed in the
#  Signum file and the rest of the commands are cat'ed
#  with it.  Sigsort2 calls the sort routines.  Currently
#  the MAXIMUM NUMBER OF SIGNALS THIS PROGRAM CAN HANDLE IS
#  >>> 100 <<<.  To increase this. duplicate one of the sigsort
#  files and modify Sigsort1 to accomodate your additions.

# Remove the last produced sortedsig file from the next
#  directory so that an error will produce a blank file
#  and not the last file.
cd ../FINALSORT
cd ../SIGSORT

# Determine the total number of signals in the input file
#  to speed the sort routine.  Only the number of signals
#  in the file (in even blocks of 10) will be searched for
#  during the sort.

awk -f numberofsig.cmd property4.dat

# Create a new command file (Sigsort2) by placing the number
#  of signals found (signum) as a variable in the last
#  routine at the top of the current Sigsort1 command file.
#  This was required because of the problem of passing a
#  variable between command files and awk routines.

cat Signum Sigsort1 > Sigsort2

#  Make the new command file executable.

chmod +x Sigsort2
Sigsort2
```

```
= Remove all the unneeded files to save space.

rm Signum
rm Sigsort2
rm *.dat sortedsig


*******************************************************************************
*******************************************************************************
```

```
# >>>>>>>>>> filename Sigsort1 <<<<<<<<<<<<<<<<
# ---------- directory SIGSORT ----------------
# ********** input file property4.dat **************
# *********** output file sortedsig **************

# This set of routines sorts the file property4.dat
#  by signal numbers and files them in order in the
#  sortedsig file.  This second command file is required
#  so that the total number of signals can be appended
#  as a variable before this command file is run.
#  After the number of signals is appended the entire file
#  is run as Sigsort2.
#  This shortens the time required for short
#  sorts.

echo "Sorting the property4.dat file for signal numbers"

echo "There are "$a " signals to be sorted"

# The sort routine can only handle 100 signals at this time.
#  This can be changed easily by adding more sort routines
#  for the higher numbers and changing this file to accomodate
#  them.

if ($a >= 99) then
echo "ERROR DETECTED...This program cannot act on"
echo "more than 99 signals.  Terminated...."
        goto end

# Based on the number of signals in the variable "a", branch
#  to the correct secton below for sort.

else if($a <= 9)then
         goto zero
else if($a <= 19) then
                goto one
else if($a <= 29) then
                goto two
else if($a <= 39) then
                goto three
else if($a <= 49) then
```

```
                goto four
else if($a <= 59) then
                goto five
else if($a <= 69) then
                goto six
else if($a <= 79) then
                goto seven
else if($a <= 89) then
                goto eight
endif

# All of these awk routines are the same except for the
#  actual numbers sought.  Awk can only handle 10 output
#  files at a time, therefore, the routines each sort for
#  10 signal numbers and place each one found in a separate
#  file.  There are a maximum of 99 files created.  This
#  is probably inefficient, but it works.  Messages on the
#  number of signals left to sort are sent to the screen.

awk -f sort90-99.cmd property4.dat
eight:
awk -f sort80-89.cmd property4.dat
seven:
echo "Less than eighty to go"
awk -f sort70-79.cmd property4.dat
six:
awk -f sort60-69.cmd property4.dat
five:
echo "Less than sixty to go"
awk -f sort50-59.cmd property4.dat
four:
awk -f sort40-49.cmd property4.dat
three:
echo "Less than forty to go"
awk -f sort30-39.cmd property4.dat
two:
awk -f sort20-29.cmd property4.dat
one:
echo "Less than twenty to go"
awk -f sort10-19.cmd property4.dat
zero:
echo "Less than ten to go"
awk -f sort1-9.cmd property4.dat
echo "the results will be in the sortedsig file"
```

```
= When done, cat all the files together.  Starting at
=  zero and sequentially add them to the file sigfile.

cat sigfile** > sortedsig

# Remove all the sigfiles to conserve space and prevent errors
#  during the next compiler run.

rm sigfile**
echo "sending sortedsig file to FINALSORT directory"
mv sortedsig ../FINALSORT/

# The end statement is a label used by the "too many signals"
#  error routine above to end the program.

end:

*******************************************************************************
*******************************************************************************
```

```
*****************************************************************************
*****************************************************************************

# >>>>>>>>>>> filename numberofsig.cmd <<<<<<<<<<<
# ---------- directory SIGSORT ----------------
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses Sigsort as output file *********
# This routine counts the number of signals in the
#  input file and then creates the file "signum" with
#  a single line entry.  It is "set a = <number of signals
#  in the input file>.  This is then concatenated at the
#  top of the command file Sigsort1 creating Sigsort2.
#  It dramatically reduces the sort time for small numbers
#  of signals.


BEGIN {FS = "\""; RS = ";";  file = "Signum"}

# Search for signal numbers not property numbers.  The
#  variable "a" will be set to the highest signal number
#  detected.

$2 ~ /[0-9]/ && $2 !~ /P/{a = $2}

# When done print to the file.  Note the "print #".  This
#  is required because a cshell command file must start
#  with the symbol #.

END {print "#" > file
    print "set a = "a > file}

*****************************************************************************
*****************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>> filename sort1-9.cmd <<<<<<<<<<<<<
# ---------+++ directory SIGSORT ------------------
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses   sigfile #1-9as output file *********
# This segment sorts by signal number into files in groups
#  of 10 signals per file (max ten files therefore currently
#  limited to 100 signals).
# The variable "size" is basically the only entry that changes
#  in all the other sort routines.  This single file also creates
#  a sigfile00 file so a title can be created.

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 0
      print ">>>>> SORTEDSIG a data FILE <<<<<<<<:" > "sigfile00"}

# Search for signal numbers in 2 different fields in numberical
#  order.  When found send it to a corresponding file.
{
for (i = size:i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile0"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile0"i}
                                    }
}

**********************************************************************
**********************************************************************
```

```
**********

**********

# <<<<<<<<<< filename sort10-19.cmd >>>>>>>>>>
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile #10-19as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 10}
{
for (i = size:i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                    }
}


**********

**********
```

```
**********************************************************************************
**********************************************************************************

# >>>>>>>>>>> filename sort20-29.cmd <<<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 20-29   as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 20}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                    }
}

**********************************************************************************
**********************************************************************************

# >>>>>>>>>>> filename sort30-39.cmd <<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses   sigfile # 30-39  as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 30}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                    }
}

**********************************************************************************
**********************************************************************************
```

```
**********************************************************************
**********************************************************************

= >>>>>>>>>> filename sort40-49.cmd <<<<<<<<<<
= ****** USE PROPERTY4.DAT AS INPUT FILE **********
= ******uses sigfile # 40-49    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\"": RS = ";": OFS = "\"";size = 40}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                }
}

**********************************************************************
**********************************************************************

# <<<<<<<<<< filename sort50-59.cmd <<<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 50-59    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\"": RS = ";": OFS = "\"";size = 50}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,":" > "sigfile"i}
                                }
}

**********************************************************************
**********************************************************************
```

```
**********************************************************************
**********************************************************************

= <<<<<<<<<< filename sort60-69.cmd <<<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 60-69    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 60}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                    }
}

**********************************************************************
**********************************************************************

# >>>>>>>>>>> filename sort70-79.cmd <<<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 70-79    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = \""; RS = ";"; OFS = "\"";size = 70}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                    }
}

**********************************************************************
**********************************************************************
```

```
**********************************************************************
**********************************************************************

= >>>>>>>>>>> filename sort80-89.cmd<<<<<<<<<<<
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 80-89    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 80}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,":" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                }
}

**********************************************************************
**********************************************************************

# >>>>>>>>>> filename sort90-99 >>>>>>>>>>>>>
# ****** USE PROPERTY4.DAT AS INPUT FILE **********
# ******uses sigfile # 90-99    as output file *********
# this segment sorts by signal number into files in groups
# of 10 signals per file (max ten files therefore currently
# limited to 100 signals).

BEGIN {FS = "\""; RS = ";"; OFS = "\"";size = 90}
{
for (i = size;i <= (size + 9);i++){
if($4 ~ /[0-9]/ && $4 == i){print $2, $3, $4,";" > "sigfile"i}
if($5 ~ /[0-9]/ && $5 == i){print $2, $3, $4, $5,";" > "sigfile"i}
                                }
}

**********************************************************************
**********************************************************************
```

```
*****************************************************************************
*****************************************************************************

#  >>>>>>>> filename Finalsort <<<<<<<<<<<<
# ++++++++++++++++ directory FINALSORT ++++++++++++
# ****** Initial input file sortedsig *********
# ******** Final output file Final.dat *********
# This command file completes the compilation of the program
# portion of the MACPITTS input code and sends it to the
# THESISDEF directory for addition of a program title, a
# definitions block and a process title.  Those additions
# complete the MACPITTS input file.

echo "determining parallel intstates"
awk -f parain1.cmd sortedsig
cat sortedsig2 sortedsig > resortedsig
echo "attaching intstate outputs to inputs"
awk -f out1.cmd resortedsig
cat resortedsig1 sortedsig2 > outin.dat
echo "intstate connections completed"
awk -f inout.cmd outin.dat
cat finalinout.dat sortedsig > yesnoin.dat
echo "connecting intstates to yes and no conditionals"
awk -f yesin1.cmd yesnoin.dat
cat yesno.dat sortedsig > acheck.dat
awk -f ayes.cmd acheck.dat
cat finalinout.dat aconnect.dat acheck.dat > headerck.dat
awk -f header.cmd headerck.dat
cat startheader.dat headers.dat > Final.dat
echo " all labels attached... sending Final.dat to "
echo " THESISDEF directory "
cp Final.dat ../THESISDEF/Final.dat
rm *.dat *ortedsi*

*****************************************************************************
*****************************************************************************
```

```
= >>>>>>>>>>>> FILE NAME parain1.cmd  <<<<<<<<<<<
# ---------- directory FINALSORT ----------
# ********* input file is sortedsig      *******
# ********* output file is sortedsig2 **********
# This segment pulls the "in" lines and attaches a flag
#  called incheck to each line.
#  ANY parallel "in" lines are tied together in the correct
#  MACPITTS format and placed in sortedsig2 ready
#  for correct "in" attachment procedures to follow.

BEGIN{i = 1; FS = "\""; OFS = "\""; file = "sortedsig2"}

# Search the second field for an "in" line and if found create
#  three arrays: x for the signal number, z for the property
#  number and y for the concatenation of all the parallel signals.
#  The index for all arrays is the signal number, therefore, the
#  y array only concatenates states or intrastates that are in parallel
#  because they have the same signal number.

$2 ~ /^in$/{x[$4] = $4
            z[$4] = $1
              y[$4] = y[$4] $3}

# When the entire input file has been processed, print out the
#  concatenated lines and add the keyword "incheck".  Include the
#  signal and property number.  All existing lines in the array
#  are printed, all blank lines are ignored by checking the
#  x array for a value.
#  The output will be be placed at the top of the sortedsig file
#  for use by the next awk routine.
END{for (a = 1;a <= 200 ;a++){if (x[a] != "")
                    print"incheck", y[a], x[a],z[a], ";"> file}}
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>>> FILENAME OUT1.CMD <<<<<<<<<<<<<<
# ----------++ directory FINALSORT +++++++++++++++
# ********* input file is resortedsig ************
# ********* output file is resortedsig1 **********
# This segment attaches the appropriate "in" command
#  to the corresponding "out" command based on signal
#  number matches.  This is used to connect go transition
#  statements to the end of the correct action statements
#  in the final MACPITTS code form.  All intrastate out's
#  must go to another state.  This go transition was
#  attached to the "incheck" lines earlier by the parain1.cmd
#  program.


BEGIN{a = 1; FS = "\""; OFS = "\""; file = "resortedsig1"}

# Search for the keyword "incheck" lines at the top of
#  the file and load the x and y array with the action statement
#  and signal number.

$1 ~ /^incheck$/{x[$3] = $2
                 y[$3] = $3}

# Now look for the "out" pin name lines (ignoring all incheck
#  lines).  If the signal number matches a signal number
#  stored in array y then print the "out" property number,
#  out, the action statement, the transition statement from
#  the corresponding x array and the signal number.

$1 !~ /^incheck$/ && $2 ~ /^out$/{
        if($3 == y[$3]){
                print$1, $2, x[$3], $3,";" > file
                     }
                       }
# This output file will be concatenated with the previously
#  generated "incheck" file for use by the next awk routine.
#  This output file must be placed above the incheck file
#  when concatenated.

**********************************************************************
**********************************************************************
```

```
# >>>>>>>>>>>> FILE NAME inout.cmd     <<<<<<<<<<
# ---------- directory FINALSORT ++++++++++++
# ********* input file is outin.dat      *******
# ********* output file is finalinout.dat*******
# This segment just ties the out pin name action to
#  the in pin name action based on the property number
#  match.  This output is now in final MACPITTS action
#  statement form.  The input file contains only the
#  "out" pin name lines and the "incheck" lines.

BEGIN{FS = "\""; OFS = "\""; file = "finalinout.dat"}

# Search for an "out" line keying on the property number in
#  field 1.  If found then create 3 arrays.  Load array x
#  with the poperty number, y with the action and z with the
#  signal number. The array indexes are the property number.
#  These arrays must be completely loaded before an "incheck"
#  line is reached.

$1 ~ /[0-9]/{x[$1] = $1
             z[$1] = $4
               y[$1] =$3}

# After the array is loaded check for the "incheck" keyword.
#  The input data file must have all the "out" lines listed
#  before these "incheck" lines to get the correct results.
#  This is an AWK limitation.
# Based on the property number of the incheck line the correct
#  action array element is concatenated with the action field
#  of the incheck line.  The result is stored in the same y
#  array element.  The z aray is updated with the new signal
#  number.  Note that the property number is still the array
#  index.  AWK allows any string to be an index.  It need not
#  be numerical.

$1 ~ /^incheck$/{x[$4] = $4
             z[$4] = $3
               y[$4] = $2 y[$4]
                 print "finalin", $4, y[$4], $3 > file}

# The result of this concatenation of actions is printed in the
```

```
= output file with a new keyword "finalin" for use in later
= routines.  The property and signal numbers of the correct
= incheck line is also included.  The complete parallel and
= non parallel action statements have now been combined with
= the "go" transition statement and are now ready for
# combination with the correct conditionals or state names.
# This file will then be attached to the top of the original
# sortedsig file for use in the next awk routine.

*******************************************************************************
*******************************************************************************
```

```
# >>>>>>>>>>>> FILE NAME yesin1.cmd    <<<<<<<<<<
# ********** input file is yesnoin.dat    ********
# ********** output file is yesno.dat  **********
# This routine searches for "yes" and "no" pin name lines
#  and attaches the appropriate action transition lines to them.
#  A new keyword "checked" is added to the output to facilitate
#  the next awk routine's search. The property number of the "yes"
#  and "no" pin names is also added to the output.  This output
#  file is now in correct MACPITTS conditional action statement form.
#  The action transition statements were constructed by the inout.cmd
#  awk routine and contain the keyword "finalin" as a search designator.


BEGIN{FS = "\""; OFS = "\""; file = "yesno.dat"}

# Search for the key word "finalin" and create 3 arrays; x is the
#  property number, y is the action transition statement and z is
#  the signal number.  The array indexes are the signal number
#  because the second search keys on a signal number match between
#  the data in the arrays and the yes and no pin name lines.

$1 ~ /^finalin$/{x[$4] = $2
                 y[$4] = $3
                 z[$4] = $4}

# After the arrays have been loaded the yes and no pin name lines
#  are searched for and the keyword plus the property number,
#  pin name, attached conditinal header and the action transition
#  statement are printed in the output file.  Note the search for
#  2 different fields.  The "no" pin name line doesn't carry the
#  conditional header because the "no" portion of a conditional
#  always follows the "yes" evaluation.  Finally, if there are
#  "checked" lines that have not been attached to a yes or no then
#  they must be included in the output file. This assures that all
#  intrastate actions are included for the final compilation
#  done in the awk routine Header.cmd.
```

```
{if ($2 == "yes"    $2 == "no"){
        if($4 == z $4 )
                { print "checked",$1, $2,$3  y $4 , ";" > file}
      else if($3 == z $3 )
         {print "checked",$1, $2, "      (t " y[$3 "]", ";" > file}
        else {print "checked",$0 > file}
                        }
}
```

****

****

```
# >>>>>>>>>>> FILE NAME AYES.CMD     <<<<<<<<<<
# ---------- directory FINALSORT ++++++++++++++
# ********* input file is acheck.dat     *******
# ********* output file is aconnect.dat*********
# This segment is used to evaluate a two level "series"
#  conditional flowchart and construct the required
#  MACPITTS equivalent. This is the start of a "case basis"
#  conditional limited to only 2 cases!!!!  A suggested
#  solution is included.
# The yes and no action transition statements are keyed to
#  the preceeding symbol in the flowchart using the signal
#  number of the "a" pin name of the conditional being
#  evaluated.  If it is found that the matching signal number
#  is connected to another "yes" or "no" pin name then a
#  sub conditonal statement is constructed and placed in the
# output file.  The order of this construction depends upon which
#  pin of the preceeding conditional the curent conditional is
#  attached to.  There are 2 separate blocks of code below to
#  handle these cases.  All required parenthesis are added here.
#  Each line of the output has the keyword "doublecond" attached
#  for use in the next awk routine.  The array "yessigname" and
#  "nosigname" contain the yes/no action transition statements
#  generated earlier and keyed by the "checked" keyword.  Their
#  index is the property number of the conditional.  This allows
#  attachment by property number to the correct "a" pin name
#  line for use in its signal name search noted above.
# Recapitulation: The yes/no action is connected to its "a" pin
#  name by property number.  The "a" signal number is compared
#  for another yes/no pin name match.
# A complete block of MACPITTS code is generated for a 2 condition
#  case basis format.  NOTE MORE THAN 2 CASE BASIS CONDITIONALS
#  CANNOT CURRENTLY BE COMPILED BY THIS PROGRAM.

# RECOMMENDED SOLUTION: The input file acheck.dat indicates a case
#  basis construct when the "no" pin name line contains a signal
#  number in its 4th field instead of a normal conditional action
#  transition statement.  (This is in a "checked" keyword line).
#  Use this fact and work top down from the state name "out" to
#  conditional "a" pin name connection and print only the property
#  number matched yes commands until a "no" line is reached that
#  has an action statement atached to it.  This will require multiple
```

```
= arrays and a printout to the file only after the entire input
= file has been evaluated. Check the "acheck.dat" and "aconnect.dat"
= files for a better understanding. To generate these, comment out
= the rm *.dat line at the end of the Finalsort command file.

BEGIN{a = 1; FS = "\""; OFS = "\""; file = "aconnect.dat"}

# Load the arrays based on the keyword "checked"

$1 ~ /^checked$/ && $3 ~ /^yes$/{yes[a] = $2
                                 yessigname[$2] = $4}
$1 ~ /^checked$/ && $3 ~ /^no$/{nosigname[$2] = $4}

# Search for an "a" pin name line and load the 2 variables.

$2 ~/^a$/{signuma = $3
          propnuma = $1}

# Check the "yes" and "no" lines for a signal number match to
# the "a" found above. This assumes that the "a" and "in" lines
# are before the yes and no lines (guaranteed in THESISPROP).
# If found print out in correct format including parenthesis
# and a "t" for the "no" line. Insert the keyword "doublecond"
# for later awk routine use.

{if (($4 == signuma) && ($2 == "yes")){
                        print $3, "doublecond", $1, ";" > file
    print "    " yessigname[propnuma], "doublecond", ";" > file
     print "    " nosigname[propnuma], "doublecond", ";" > file
                    print  nosigname[$1] ")", "doublecond", ";" > file
                    print "signalend","", $1, ";" > file}
}
{if (($3 == signuma) && ($2 == "no")){
                print yessigname[$1],"doublecond",$1 > file
          print "   (t " yessigname[propnuma], "doublecond", ";"  > file
     print "    " nosigname[propnuma] ")", "doublecond", ";"  > file
                    print "signalend","", $1, ";"  > file}
}

*************************************************************************
*************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>>> FILE NAME HEADER.CMD  <<<<<<<<<<
# ---------- directory FINALSORT ++++++++++++++
# ********* input file is headerck.dat  *******
# ****** output file is startheader.dat, headers.dat *******
# This is currently the most complex routine in this compiler.
#  It assembles all MACPITTS formated code under the correct
#  state name based on 2 types of states; the first state in
#  a process and all other states in a process.
# Three blocks of arrays are first loaded and contain the
#  doublecond blocks, the single condition lines and the
#  action only lines (no conditional test involved in the
#  action determination).  The number of conditonals in
#  parallel (number of "a's" in parallel) is then determined
#  and the corresponding signal and property numbers are stored
#  in another set of arrays.
# Two searches are now started for the "start" state title or
#  name and all "other" state titles.  When one is found, 3
#  possible blocks of MACPITTS code are appended to the state
#  name and sent to the correct output file.  The order of
#  the the check for this block code is critical.  The first
#  check looks for actions involving no conditionals.  If none
#  are found then the existance of a "series" or dual case
#  conditional is checked.  Finally, if the previous checks
#  failed then the single conditional is checked.  The reason
#  for this order is that all the data used to construct the
#  single and dual conditonals earlier is still available and
#  checking out of sequence would provide an incomplete code
#  bulidup.
# The result of these checks is printed in the correct output
#  file.  Note that the "start" state file will only have 1
#  entry while the "other" state file may have many.  Since
#  the order of execution of these states in MACPITTS depends
#  on the transition designated in each action statement (except
#  for the initial state) their order in this file is unimportant.

BEGIN{numparallel = 1;a = 1; b = 1; parenflag = 0; FS = "\"";
 OFS = "\""; file1 = "headers.dat":file = "startheader.dat"}

# Setup and load the doublecond, singlecond, and no conditional
#  arrays.  Note that the doublecond array has a start and
#  stop indicator to break up the array in sequential blocks.
```

```
$1 ~  ^finalin$ {finalprop b  = $2
                finalword b  = $3
                b--}
$2 ~ /^doublecond$ {dblprpnum[$3] = $3
                        dblsigname[a] = $1
                    startnum[$3] = a++}
$1 ~ /^signalend$/{stopnum[$3] = a}
$1 ~ /^checked$/ && $3 ~ /^yes$/{yes[$2] = $2
                                 yessigname[$2] = $4}
$1 ~ /^checked$/ && $3 ~ /^no$/{no[$2] = $2
                                nosigname[$2] = $4}
# Check for conditionals in parallel, note the number
#  in parallel and their signal and propertynumbers.
#  Note that non conditional actions are not checked
#  for parallel status because they have already been
#  compiled in parallel format (remember all conditional
#  actions are accomplished in parallel and the non conditional
#  actions follow the same format).

$2 ~/^a$/{if($3 == signuma[numparallel]){
                                    numparallel++
                                    signuma[numparallel] = $3
                                    propnuma[numparallel] = $1}
        else{numparallel = 1
            signuma[numparallel] = $3
           propnuma[numparallel] = $1}
        }
# Set up the variables for a non-conditional action check.

$2 ~ /^in$/{prev = NR
          inprop = $1
          insignum = $4}
# Search for a start state title and if found make the 3 checks
#  previously noted in the explanation above.  Note again that
#  the sequential ordering of these checks is critical.

$2 ~ /^outstart$/{print $3 > file
       print "  (par" > file
        {if((NR == (prev + 1)) && ($4 == insignum)){
         for(k = 1; k <= b; k++){if(finalprop[k] == inprop)
                                print "     " finalword[k] > file
                                  parenflag = 1}}}
# The parenflag above is required to signal that this non-conditional
#  action printout already has the required number of parenthesis.  The
```

```
= conditional blocks below require more parenthesis to complete the
= block.

 for(j = 1; j <= numparallel; j--)
  {if(($4 == signuma[j]) && (dblprpnum[propnuma[j]] == propnuma[j])){
        for (i = startnum[propnuma[j]]; i <= stopnum[propnuma[j]];i--){
                print "    " dblsigname[i] > file }}
    else if(($4 == signuma[j]) && (yes[propnuma[j]] == propnuma[j])){
      print "     " yessigname[propnuma[j]] > file
        print "     " nosigname[propnuma[j]] > file
        print  nosigname[$1] > file }}
      if(parenflag != 1){ print "  )" > file}
    else{parenflag = 0}}

# Do the same as above. but check for state names of states other
# than the start state.

$2 ~ /^outother$/{print $3 > file1
        print "  (par" > file1
        {if((NR == (prev + 1)) && ($4 == insignum)){
          for(k = 1; k <= b; k--){if(finalprop[k] == inprop)
                print "     " finalword[k] > file1
                  parenflag = 1}}}
        for(j = 1; j <= numparallel; j++)
        {if(($4 == signuma[j]) && (dblprpnum[propnuma[j]] == propnuma[j])){
           for (i = startnum[propnuma[j]]; i <= stopnum[propnuma[j]];i++){
        print "    " dblsigname[i] > file1 }}
        else if(($4 == signuma[j]) && (yes[propnuma[j]] == propnuma[j])){
           print "     " yessigname[propnuma[j]] > file1
           print "     " nosigname[propnuma[j]] > file1
           print  nosigname[$1] > file1 }}
     if(parenflag != 1){ print "  )" > file1}
        else{parenflag = 0}}
# MODIFICATION OF THIS ROUTINE MAY BE REQUIRED IF CHANGES ARE MADE
# TO THE AYES.CMD FILE TO HANDLE MULTIPLE CASE BASIS FLOWCHARTS.
# RECOMMENDED SOLUTION: Add any change to the last two blocks of
# code between the non-condition search and the doublecond search
# (just below the parenflag = 1 lines). Note that the printing of "(par"
# must also be deleted in this case, therefore, this print command
# will have to be moved within the action blocks of each of the
# checks. Loading of any arrays
# can be accomplished anywhere within the current block of array
# loading routines. Ensure that any keyword lines that have been
# created for this capability are included in the file above the
```

```
= acheck.dat file when concatenating.  See the command file Finalsort
= for all current concatenations.

*****************************************************************************
*****************************************************************************
```

```
*****************************************************************************
*****************************************************************************

# >>>>>>>>> Command file Thesisdef <<<<<<<<<<
# ---------- directory THESISDEF ---------------
# ******* initial input file Final.dat, td *******
# ****** final output file <filename>.mac *******
# This command file completes the MACPITTS input code.  It generates
#  the program title line, process line, and definitions and then
#  concatenates all the required files in the correct sequence
#  for a single process MACPITTS input file.

echo " "
echo "Please enter the desired output file name "
echo -n "do not use a . extension.  A .mac extension will be added :"
echo " "
set a = $<
echo "The outfile name will be $a.mac"
awk -f thesis1def.cmd td
cat def1.dat >> def.dat
awk -f thesis3def.cmd def.dat
awk -f thesis4def.cmd def.dat
cat title.dat finaldef.dat signaldef.dat process.dat Final.dat > $a".mac"
echo "))" >> $a".mac"
echo " "
echo "COMPLETED FILE IS IN" $a".mac IN THE MAIN DIRECTORY"
echo " "
mv $a".mac" ../../$a".mac"
rm *.dat sortedsig td

*****************************************************************************
*****************************************************************************
```

```
*******************************************************************************
*******************************************************************************

# >>>>>>>> filename thesisdef1.cmd <<<<<<<<<<<<<<
# +++++++++ directory THESISDEF ++++++++++++++++
# ********* input file td ************************
# ********* output file def.dat, def1.dat *******
# This routine sets up 2 files; one with the program title
# and process identifiers and and the other with all the
#  signal numbers and names.  These two files will be
#  concatenated together (def.dat on top) for use by the
#  next awk routine.  That routine will tie definitions
#  with signal names and the title and process information
#  with its information contained in the signal names.

BEGIN {FS = "\"";RS = ";"; OFS = "\""; file = "def.dat";
      file1 = "def1.dat"; print "DEF.DAT_FILE;" > file}

# Pull the lines that are signals, def's or title/process
#  lines and place in the appropriate file.

$3 ~ /^def$/ && $4 !~ /^0$/{print $3, $4, ";" > file}
$3 ~ /^title$/ || $3 ~ /^process$/ {print $3, $4, ";" > file}
$2 ~ /[0-9]/{print $2, $3, ";" > file1}

*******************************************************************************
*******************************************************************************
```

```
**********************************************************************
**********************************************************************

# >>>>>>>>>>>> file name thesis3def.cmd <<<<<<<<<<<<<<<<<<<<
# ------------ directory THESISDEF ----------------------------
# ********** input file name def.dat (not needed )*******
# ********** output file name finaldef.dat **************
# This routine just creates the ground and 3 phase clock
# definitions.  This could have been done in a variety of ways,
# this was the easiest at the time.  It is used to assure
# regeneration of this data file should it accidently be
# removed or modified.

BEGIN {FS = "\""; RS = ";"; file = "finaldef.dat";
                        print "(def 1 ground)" > file
                        print "(def 2 phia)" > file
                        print "(def 3 phib)" > file
                        print "(def 4 phic)" > file}


**********************************************************************
**********************************************************************
```

```
# >>>>>>>>>>> file name thesis4def.cmd <<<<<<<<<<<<<<<<<<<
# ---------- directory THESISDEF ----------
# *********** input file def.dat ************************
# *** output file title.dat, process.dat signaldef.dat **
# This routine designates the signal numbers that correspond
#  to the title, process, and MACPITTS definitions.  It then
#  creates the MACPITTS program title line, process line and
#  all definition lines.  The three files will be concatenated
#  with the other files to create the final MACPITTS input
#  code.
#  The current concatenation order is title.dat, finaldef.dat
#  signaldef.dat process.dat and Final.dat as shown in the
#  thesisdef.cmd file.

BEGIN {FS = "\""; file = "title.dat"; file1 = "process.dat";
     file2 = "signaldef.dat"}
$1 ~ /^def$/{def[$2] = $2}
$1 ~ /^title$/{title = $2}
$1 ~ /^process$/{process = $2}
{if ($1 == def[$1]) print "(def", $2")" > file2}
{if ($1 == title){print "(program",$2 > file
             print "" > file}}
{if ($1 == process){ print "" > file1
                print "(process",$2 > file1
                print "" > file1}}
```

LIST OF REFERENCES

1. Valid Logic Systems Incorporated. **SCALD System Reference Manual.** Document Number 900-00016. 1984.

2. Southard, J.R., **An Introduction MACPITTS.** Defense Advance Research Project Agency under Electronic System Division Control, Lincoln Labs M.I.T, Document Number SI 962880C0001, 1983.

3. Southard, J.R., **MACPITTS an Approach Silicon Compilation.** Lincoln Labs M.I.T, Dec 1983.

4. Carlson, Dennis J., **Application of a Silicon Compiler VLSI Design of Digital Pipelined Multipliers.** Master's Thesis, Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, California, June 1984.

5. Aho, A.V., **AWK – A Pattern Scanning and Processing Language.** Bell Laboratories, Murray Hills, N. J., Sept 1978.

6. Stone, Harold S., **Introduction Computer Architecture.** Science Research Associates Incorporated, Chicago, Ill., 1975.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5000	2
3.	Department Chairman, Code 62 Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
5.	Dr. D.E. Kirk, Code 62Ki Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	5
6.	Dr. H.H. Loomis Jr., Code 62Lm Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5000	2
7.	LCDR J. Harmon, USN SMC #2231 Naval Postgraduate School Monterey, California 93943-5012	1
8.	LTJG. A Mullarky, USN SMC #1424 Naval Postgraduate School Monterey, California 93943-5012	1

9.	LCDR M.A. Malagon-Fajar, USN 1220 7th Street, # 2 Monterey, California 93940	1
10.	Major E.L. Weist Jr., USMC 3012 Bayer Dr. Marina, California 93933	1
11.	Mr. Doug McCafferty Senior Sales Engineer VALID 4699 Old Ironsides Drive, Suite 150 Santa Clara, Ca 95050	1
12.	Capt E.G. Malagon, USMC 1220 7th Street, # 2 Monterey, California 93940	1
13.	Prof. R. McGhee, Code 52Mz Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
14.	Mr. P. Blankenship Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1
15.	Mr. J. O'Leary Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1
16.	Mr. A. Casavant Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1

17. Dr. T. Bestul 1
Naval Research Laboratories
Code 7590
Washington D.C. 20375

18070 2

21941T

Thesis
W3835
c.1

Weist
A flowcharting system
and compiler interface
for MACPITTS.

21941T

Thesis
W3835
c.1

Weist
A flowcharting system
and compiler interface
for MACPITTS.

DUDLEY KNOX LIBRARY
3 2768 00018076 4